AFRL-IF-WP-TR-2004-1568

COMPONENT COMPOSITION FOR EMBEDDED SYSTEMS USING SEMANTIC ASPECT-ORIENTED PROGRAMMING

Dr. Martin Rinard

Massachusetts Institute of Technology
Laboratory for Computer and Science
77 Massachusetts Avenue
Cambridge MA 02139-4307

OCTOBER 2004

Final Report for 12 June 2000 – 12 August 2004

STINFO FINAL REPORT

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

NOTICE

/s/
MARVIN M. SORAYA
Project Engineer
Advanced Architecture &
Integration Branch

/s/
STEPHEN L. BENNING
Team Lead
Advanced Architecture &
Integration Branch

/s/
DAVID A. ZANN, Chief
Advanced Architecture &
Integration Branch
Materials & Manufacturing Directorate

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YY)* October 2004	**2. REPORT TYPE** Final	**3. DATES COVERED *(From - To)*** 06/12/2000 – 08/12/2004

4. TITLE AND SUBTITLE COMPONENT COMPOSITION FOR EMBEDDED SYSTEMS USING SEMANTIC ASPECT-ORIENTED PROGRAMMING	**5a. CONTRACT NUMBER** F33615-00-C-1692
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 69199F
6. AUTHOR(S) Dr. Martin Rinard	**5d. PROJECT NUMBER** ARPI
	5e. TASK NUMBER FS
	5f. WORK UNIT NUMBER 0K
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Massachusetts Institute of Technology Laboratory for Computer and Science 77 Massachusetts Avenue Cambridge, MA 02139-4307	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Information Directorate Air Force Research Laboratory Air Force Materiel Command Wright-Patterson AFB, OH 45433-7334	**10. SPONSORING/MONITORING AGENCY ACRONYM(S)** AFRL/IFSC
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S) AFRL-IF-WP-TR-2004-1568

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The goal of our research was to develop technologies and techniques in support of real-time systems for the defense community. Our research focused on Real-Time Java implementation and analysis techniques. Real-Time Java is important for the defense community because it holds out the promise of enabling developers to apply COTS Java technology to specialized military embedded systems. It also promises to allow the defense community to utilize a large Java-literate workforce for building defense systems.

Our research has delivered several techniques that may make Real-Time Java a better platform for developing embedded systems. These techniques include ways to implement scoped memories (a key Real-Time Java construct) without the possibility of introducing unexpected and potentially catastrophic delays in the execution of real-time threads, analyses that ensure the correct use of Real-Time Java scoped memories, analyses that compute how much memory is required to execute a given Real-Time Java program (potentially helping developers calculate how much memory must be including in a given system to ensure that the system will execute without running out of memory), and optimizations that reduce the amount of memory required to execute a Real-Time Java program.

15. SUBJECT TERMS
Real-Time Systems, Embedded Systems, Real-Time Java, Scoped Memory, Object-Oriented Programming, Instrumented Semantics

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT:**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** (Monitor)
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	SAR	270	Marvin Soraya
Unclassified	Unclassified	Unclassified			**19b. TELEPHONE NUMBER** *(Include Area Code)* (937) 255-4709 x3177

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

Contents

Chapter 1

Executive Overview

1.1 Introduction

The focus of our research during this project was the analysis and implementation technologies for the Real-Time Specification for Java (RTSJ), a standard extension to Java for real-time systems [38]. The motivation for this focus was the difficulty of developing critical real-time systems for the defense community using standard existing development methodologies and the need for the defense community to track modern software development technologies more closely (both to take advantage of improvements and to help ensure the availability of a suitably trained workforce). At the same time, the Department of Defense has special needs that the broader COTS community will not serve on its own. Real-Time Java holds out the promise of providing a solution that is largely based on and tracks COTS technology but is enhanced with features that make it suitable for building large and complex defense systems. Our research goal was to develop key technology that would promote the ability of the defense community to use Real-Time Java more effectively.

Our research produced results in several broad areas: analyses and implementation techniques for scoped memories in Real-Time Java, analyses and optimizations for reducing the amount of memory required to run Real-Time Java programs, and analyses for tracking the conceptual roles that objects play in Real-Time Java programs. All of this research has been published over the course of the project. We have also developed prototype implementations of many of our algorithms in the MIT FLEX compiler infrastructure, which is freely available over the Internet.

Highlights of our specific activities and accomplishments included:

- An analysis for ensuring the safety of Real-Time Java programs that use scoped memories. Scoped memories are a key element of Real-Time Java, but must be used correctly to avoid the possibility of dynamic exceptions which can cause the program to fail or behave unpredictably. Our analysis checks the program to verify that it is free of any such errors.

- An example scenario for how it would be useful is a UAV (Unmanned Airborne Vehicle) with a feed coming in to an automatic target recognition component.

Without our analysis, the component might have a software error that would cause the system to fail, losing video or target recognition capability. Our software would find such an error and enable the developer to eliminate it, enabling the system to operate without the possibility of such errors.

- A real-time scheduling interface for Real-Time Java programs. This interface lets developers easily implement their own real-time scheduling algorithms, in particular scheduling algorithms that are best suited for their particular application. Without this capability a developer would have to rely on the standard scheduling algorithms provided by the system.

- An example scenario for how this would be useful is a UAV feed with automatic target recognition software and special scheduling needs. Without this interface, the developer would be forced to rely on the standard scheduling algorithm, which could suffer from suboptimal performance such a jitter problems which might make it difficult to correctly view and interpret the UAV feed. With our technology, the developer could implement their own scheduling algorithm, eliminate the suboptimal performance, and get better comprehensibility of the UAV feed.

- An analysis that automatically reduces the amount of space required to execute Real-Time Java programs. This analysis determines when it is possible to reduce the amount of bits required to represent Java objects, enabling a reduction in the amount of memory deployed in the embedded system.

  An example scenario for how this would be useful is a space reduction that would make it more practical to place ATR components on the UAV instead of on the ground, reducing required bandwidth by enabling ATR software to filter uninteresting data without transmitting it.

1.2 Real-Time Java Scoped Memories

Memory management is an important issue in Real-Time Java. Safe memory management has usually been implemented by garbage collection. But garbage collection is widely viewed as unsuitable for real-time systems because the pauses characteristic of garbage collection may perturb the execution of the system to the point that it fails to satisfy its real-time scheduling requirements.

Real-Time Java avoids this problem by using scoped memories. The basic idea is that a part of the execution allocates all of its objects in a specific scoped memory. That scoped memory is deallocated as a unit when the part finishes, without the potentially unbounded pause times characteristic of general garbage collection. The scoped memories are arranged into a hierarchy, with the lifetimes of scoped memories higher in the hierarchy containing the lifetimes of the scoped memories lower in the hierarchy.

For this approach to work, it must be the case that there are no references pointing into the scope memory when the scoped memory is deallocated. This is accomplished in the Real-Time Java spec by inserting dynamic checks into the program at every point where the program might generate a pointer from a higher scoped memory to a lower scoped memory. If the program attempts to create such a pointer, the JVM throws an exception.

1.2.1 Scoped Memory Implementation

As part of our research activities we developed an implementation of scoped memories for RTSJ. To our knowledge, this implementation was the first RTSJ scoped memory implementation ever developed. As part of this activity, we pioneered key implementation techniques and uncovered some quite subtle implementation issues. A key issue is ensuring that the scoped memory implementation does not interact at all with the garbage collector. Such an interaction could lead to unexpected pauses of unbounded duration, which would, in turn, cause the system to miss crucial real-time deadlines.

The Real-Time Specification for Java was designed to allow the scoped memory implementation to not have to interact at all with the garbage collector. We found that while it is possible to build such a scoped memory implementation, it is not completely straightforward to do so — there are several subtle points that need to be addressed to ensure the complete lack of interaction between the two memory managers.

Most of these subtle interactions take place in the context of the implementation of no-heap real-time threads. No-heap real-time threads, as the name suggests, are threads that have real-time requirements and must never interact with the garbage collector.

One potential interaction occurs when the garbage collector scans a scoped memory area looking for references at the same time as the no-heap real-time thread allocates an object in that memory area. The actions of the garbage collector must not delay the object allocation, eliminating the possibility of using locks to manage the interactions between the collector and the no-heap real-time thread. Another potential interaction occurs when a no-heap real-time thread and a normal thread share a memory area. There is a need for a lock-free synchronization mechanism that the two threads can use when they allocate memory concurrently in that region.

Our solutions to these problems rely largely on lock-free synchronization mechanisms such as compare and swap to avoid the need for blocking synchronization between no-heap real-time threads, other threads, and the garbage collector. Our algorithms are described further in reference [30].

Eliminating implementation interactions between the garbage collector and the scoped memory implementation is crucial for ensuring that the real-time threads continue to make their deadlines. Failures to meet deadlines can cause catastrophic, unpredictable failures.

1.2.2 Scoped Memory Analysis

The RTSJ specifies that the implementation must check for the absence of references from one scoped memory to a scoped memory whose lifetime is included in the lifetime of the first scoped memory. These checks are designed to ensure the absence of references into each scoped memory when the scoped memory is deallocated. If the implementation allowed such references, the system could access memory that has been deallocated, then reallocated to hold different objects. These kinds of errors are notorious for causing subtle, non-deterministic and catastrophic errors.

While the dynamic checks are a huge improvement over the alternative (dangling references with the possiblity of catastrophic errors), they still can cause the program to fail. If the program fails a dynamic check, it must throw an exception. Developers typically write code that responds to exceptions by terminating the execution or taking some other general action that is not what the system would optimally do. Of course, such a failure can potentially be very dangerous to a person using the program. If, for example, the program is part of an image pipeline feeding images to an operator for decisions (potentially as part of a UAV scenario), such a failure could cause the flow of images to stop completely, leaving the operator with no information whatsoever. If the program is controlling entities in the physical world (such as a part of a vehicle control program), the failure could leave the entities running out of control and unable to respond or correctly process inputs or commands.

Our research addressed this question by developing analysis algorithms and type systems that ensure the correct use of scoped memory areas in Real-Time Java programs. Our automatic analysis uses escape analysis to verify the correct use of scoped memories. If the analysis succeeds, it has guaranteed that the program uses scoped memories correctly. We have also developed a type system that allows the programmer to add additional region type information to a Real-Time Java program, and a type checker could check to ensure correctness statically. The program can then be translated to a Real-Time Java program that uses memory areas without generating any runtime exceptions.

Since the Real-Time Java program has been proven to use memory correctly, all checks can be removed in the Real-Time Java runtime. This not only improves the runtime performance of Real-Time Java programs, it improves safety. Specifically, there is a guarantee that the program will never fail because of a violated safety check — in other words, there is an entire class of errors that the analysis has verified will never occur in any circumstances whatsoever. This kind of verification can, in turn, eliminate potentially serious errors that can cause safety violations of the kind described above.

Using a type-safe front end also relieves the Real-Time Java runtime of the burden of the correctness of safety checks. Without the burden of implementing correct runtime safety checks, the development time of a working Real-Time Java VM can be shortened substantially.

Verification and validation is another important potential application of this analysis. Any realistic validation and verification effort would need to address the potential issue of scoped memory reference errors. Our static analysis could help direct the attention of the validation and verification effort to any potential problems and, in some cases, eliminate the need to consider this issue at all during verification and validation.

One important aspect of our project is that the analysis works for multithreaded programs. Many real-time programs contain multiple threads and any analysis for these programs must take threads into account otherwise they may produce results that are simply incorrect for multithreaded programs. The potential effect is quite negative — an incorrect analysis result can lead to an incorrect understanding or transformation of the program, with catastrophic results when the program is actually deployed. The fact that our analysis is sound for multithreaded programs is a necessary prerequisite for using it in the context of Real-Time Java, which anticipates the widespread use of threads.

More information on this research is available in the following publications: [47, 198, 191].

1.3 Real-Time Scheduling

Our Real-Time Scheduling work focused on primitives for supporting the development of real-time schedulers. The issue is that most systems provide a standard set of real-time scheduling algorithms and it can be very difficult to implement another algorithm. This is a problem since there are a large range of scheduling algorithms that would be useful if it were possible to deploy them with a reasonable effort. Our research in this area provided a new set of primitives that developers can use to easily develop their real-time scheduling algorithms.

The basic problem with previous implementation support for real-time schedulers was that the developer was essentially forced to work within the operating system kernel to develop a new scheduling algorithm. This is a very daunting task since it requires the developer to have detailed knowledge of the operating system, an area of expertise well outside that of most developers of real-time scheduling algorithms. Our interface provides developers of real-time schedulers with the functionality they need without any requirement that they write low-level operating system code. This functionality makes it possible to develop “pluggable schedulers” that can be deployed as necessary into the system for specific needs. In effect, one can view each scheduler as an aspect that our system enables to be woven easily into the system. The scheduler itself can affect the timing of the entire system and determine whether or not the system as a whole meets its goals. Integrating a new scheduler into the system however, does not require the rest of the system to be changed, a key hallmark of aspect-oriented design. See [92] for more information.

1.4 Data Size Prediction and Optimizations

Memory usage is a critical concern for embedded systems. In general, Real-Time Java programs have many sources of potential space savings. Our memory usage research focuses on two aspects: predicting the amount of memory required to execute a given program, and reducing the amount of memory required to store the program data, specifically the Java objects used to represent the data.

1.4.1 Unitary Allocation Sites

One of our mechanisms focuses on finding *unitary* allocation sites, or allocation sites for which at most one object is live at any point during the execution of the program. We have developed a static program analysis designed to find pairs of *compatible* allocation sites; two sites are compatible if no object allocated at one site may be live at the same time as any object allocated at the other site. If an allocation site is compatible with itself (these are the unitary allocation sites), then at any time during the execution of the program, there is at most one live object that was allocated at that site. It is therefore possible to statically preallocate a fixed amount of space for that allocation site, then use that space to hold all objects allocated at that site. Any further space usage analyses can then focus only on the non-unitary allocation sites.

We have also used techniques inspired from register allocation to reduce the amount of memory required to hold objects allocated at unitary allocation sites. The basic approach is to build and color an *incompatibility graph.* The nodes in this graph are the unitary allocation sites. There is an undirected edge between two nodes if the nodes are not compatible. The analysis applies a coloring algorithm that assigns a minimal number of colors to the graph nodes subject to the constraint that incompatible nodes have different colors. This information enables the compiler to statically preallocate a fixed amount of memory for each color. At each unitary allocation site, the generated code bypasses the standard dynamic allocation mechanism and instead simply returns a pointer to the start of the statically preallocated memory for that allocation site's color.

Results from our implemented analysis show that, for our set of Java benchmark programs, our analysis is able to identify 60% of all allocation sites in the program as unitary allocation sites. Furthermore, our incompatibility graph coloring algorithm delivers a 95% reduction in the amount of memory required to store objects allocated at these unitary allocation sites. We attribute the high percentage of unitary allocation sites to specific object usage patterns characteristic of Java programs: many unitary allocation sites allocate exception, string buffer, or iterator objects. See [104] for more information.

This is important for real-time systems since many real-time systems control safety-critical aspects of systems and failure can cause significant damage and threaten human lives. By helping to rule out some sources of failure and making it simpler to calculate the amount of memory required to execute the program, this analysis can help make real-time programs more reliable and make it easier to validate and verify that the program performs as expected.

1.4.2 Data Size Reductions

We have developed a set of techniques for reducing the amount of space required to hold objects in Java programs. We attack two basic sources of waste: waste in the fields (such as the class pointer and lock header) inserted automatically by the implementation, and waste in the fields inserted to represent user data. For the user data fields, we have implemented a value-flow analysis that determines the largest and smallest possible values and allocates only enough bits to hold those values. We have also implemented a variety of other analyses and transformations to reduce the total amount of memory required to execute the program. This is of importance for embedded real-time systems because it can reduce the amount of memory required to execute the program and therefore reduce the cost of the embedded real-time system. See [19] for more information.

1.5 Role Analysis

Role analysis is designed to help identify the conceptual roles that different objects play in the computation. It is useful for ensuring that the program respects many different safety properties. We have developed role analysis, which allows the programmer to state expectations about the conceptual roles that objects play in the computation. This role includes the referencing relationships of the object with other objects in the system, which allows the role system to capture important pointer information. We have developed a system for automatically extracting roles from program executions and for role checking a program that contains role annotations. For more information see [75, 140].

This information can potentially be of considerable use during validation and verification because it can make the operation of the program much more transparent to anyone attempting to reason about the program. The role extraction research is designed, in part, to help developers understand the operation of the program better and in this capacity can also support validation and verification efforts.

1.6 OEP Interaction Activities

We also worked on a variety of activities that were designed to support the program. These activities included development of the JavaCar (a first live source of video data) and the Automatic Target Recognition software component. These components helped test, evaluate, and demonstrate the technology developed in the PCES program.

The PCES program wound up centered around two Open Experimental Platforms (OEPs) — software systems that allowed groups to demonstrate their technology in various ways. PCES had a platform provided by Bolt, Beranek, and Newman (BBN) and a platform provided by Boeing. The JavaCar provided the first live video input for the BBN OEP and was instrumental in helping drive the development of the system to support external video sources. It also was important in illustrating the early viability of the platform in processing live data as opposed to recorded feeds.

The Automatic Target Recognition software was a key component of the BBN OEP for much of the project. It enabled the OEP team to demonstrate that Real-Time Java components could be successfully integrated into the OEP. It also served as an important benchmark during the project to help evaluate a variety of Real-Time Java issues including performance and demonstrated the addition of automatic image recognition capabilities into the OEP. All of these activities helped develop or demonstrate the BBN as a viable collaboration platform.

We worked extensively with the other OEP participants to coordinate the integration of these components into the OEP.

1.7 Applicability to PCES

All of this research is applicable to the basic PCES mission of better real-time software for defense applications. Our Real-Time Java scoped memory analysis and implementation research produced technology that should help Real-Time Java developers and language implementors deal more effectively with the potential issues that scoped memories raise (efficiency, unexpected exceptions). Our real-time scheduling research produced technology that may make it much easier to implement pluggable schedulers for Real-Time Java programs, which would allow developers to deploy their own custom schedulers that work well for their own applications. Our data size prediction and reduction techniques should reduce the amount of memory required to execute Real-Time Java programs and increase the reliability with which the developer can predict the amount of memory that the Real-Time Java program will need. Finally, the concept of roles and role analysis can help developers better conceptualize their proposed software structures and verify that the program does, in fact, correctly preserve those structures.

The remainder of this report integrates papers that summarize various aspects of the research. Specifically, Chapters 2 and 3 summarize our research into optimizing and analyzing memory usage. This research holds out the promise of reducing the cost of embedded systems in two ways: by making it easier to estimate the total memory requirements of the system and by reducing the amount of memory required to store the data. It can also improve the safety of the system by reducing the likelihood that the system will fail because of lack of memory - in one case because the analysis rules out many possible sources of memory usage, in others because the analysis and transformation can eliminate excess memory usage. This is important because lack of memory or an incorrect calculation of the amount of memory required to execute the program can cause the program to fail unpredictably, denying the functionality to the user of the program. For example, a video processing program could immediately terminate if it unexpectedly ran out of memory.

Chapter 4 summarizes our pointer and escape analysis for multithreaded programs. This analysis can be useful for Real-Time Java programs with threads. Its benefits (ensuring safety, eliminating check overhead) have been discussed above.

Chapters 5 and 6 discuss some of our experience with roles, a concept for helping to ensure the consistency of data structures in programs. The goal is once again to eliminate undesirable errors and unpredictable failures.

Chapter 7 discusses a type system for multithreaded Real-Time Java programs. The idea is to allow the developer more control over how the memory is managed. The goal is to allow maximum control over the allocation in Real-Time Java scoped memories while preserving safety. Chapter 8 details an algorithm for maintaining much of the advantages of a full analysis while performing only a fraction of the analysis. All of these papers are available on the Internet at www.cag.csail.mit.edu/r̃inard/paper.

1.8 Flex and Components

Flex enables the compilation, analysis, and optimization of Real-Time Java components. In its primary usage mode it is therefore basically neutral with respect to components. Various parts of the analyses in Flex could, however, substantially improve the ability of the developer to reason about the behavior of the components they compile with Flex. Moreover, Flex has been shown to be useful for processing and analyzing components in the context of the BBN OEP. Flex also served as the platform for much of the research performed as part of this contract, see, for example [31, 20].

We are delivering the Flex compiler infrastructure software on a CD.

1.9 Acknowledgements

The research for this contract was performed, in part, by a variety of MIT researchers and visitors including: Brian Demsky, Darko Marinov, Karen Zee, William S. Beebee, Jr., Cristian Cadar, Daniel Dumitran, Daniel Roy, Tudor Leu, Alexandru Salcianu, C. Scott Ananian, Suhabe Bugrara, Patrick Lam, Viktor Kuncak, Maria-Cristina Marinescu, Chandrasekhar Boyapati, Jianjun Zhao, Frederic Vivien, Robert Lee, Daniel Jackson, and Ovidiu Gheorghioiu. Collaborators outside MIT included Wei-Ngan Chin, Florin Craciun, Shengchao Qin, Sharooz Feizabadi, Binoy Ravindran, and Peng Li.

THIS PAGE WAS INTENTIONALLY LEFT BLANK

Chapter 2

Interprocedural Compatibility Analysis for Static Object Preallocation

2.1 Introduction

Modern object-oriented languages such as Java present a clean and simple memory model: conceptually, all objects are allocated in a garbage-collected heap. While this abstraction simplifies many aspects of the program development, it can complicate the calculation of an accurate upper bound on the amount of memory required to execute the program. Scenarios in which this upper bound is especially important include the development of programs for embedded systems with hard limits on the amount of available memory and the estimation of scoped memory sizes for real-time threads that allocate objects in sized scoped memories [38].

This chapter presents a static program analysis designed to find pairs of *compatible* allocation sites; two sites are compatible if no object allocated at one site may be live at the same time as any object allocated at the other site. If an allocation site is compatible with itself (we call such allocation sites *unitary* allocation sites), then at any time during the execution of the program, there is at most one live object that was allocated at that site. It is therefore possible to statically preallocate a fixed amount of space for that allocation site, then use that space to hold all objects allocated at that site. Any further space usage analyses can then focus only on the non-unitary allocation sites.

Our analysis uses techniques inspired from register allocation [7, 22] to reduce the amount of memory required to hold objects allocated at unitary allocation sites. The basic approach is to build and color an *incompatibility graph*. The nodes in this graph are the unitary allocation sites. There is an undirected edge between two nodes if the nodes are not compatible. The analysis applies a coloring algorithm that assigns a minimal number of colors to the graph nodes subject to the constraint that incompatible nodes have different colors. This information enables the compiler to statically preallocate a fixed amount of memory for each color. At each unitary allocation site, the generated code bypasses the standard dynamic allocation mechanism and instead simply returns a pointer to the start of the statically preallocated memory for that allocation site's color. The object is stored in this memory for the duration of its lifetime in the computation. Our algorithm therefore enables objects allocated at compatible allocation sites to share the same memory.

Results from our implemented analysis show that, for our set of Java benchmark programs, our analysis is able to identify 60% of all allocation sites in the program as unitary allocation sites. Furthermore, our incompatibility graph coloring algorithm delivers a 95% reduction in the amount of memory required to store objects allocated at these unitary allocation sites. We attribute the high percentage of unitary allocation sites to specific object usage patterns characteristic of Java programs: many unitary allocation sites allocate exception, string buffer, or iterator objects.

We identify two potential benefits of our analysis. First, it can be used to simplify a computation of the amount of memory required to execute a given program. We have implemented a memory requirements analysis that, when possible, computes a symbolic mathematical expression for this amount of memory [103]. Our results from [103] show that preceding the memory requirements analysis with the analysis presented in this chapter, then using the results to compute the memory requirements of unitary sites separately, can significantly improve both the precision and the efficiency of the subsequent memory requirements analysis. The second potential benefit is a reduction in the memory management overhead. By enabling the compiler to convert heap allocation to static allocation, our analysis can reduce the amount of time required to allocate and reclaim memory.

This chapter makes the following contributions:

- **Object Liveness Analysis:** It presents a compositional and interprocedural object liveness analysis that conservatively estimates the set of objects that are live at each program point.

- **Compatibility Analysis:** It presents a compositional and interprocedural analysis that finds sets of compatible allocation sites. All objects allocated at sites in each such set can share the same statically preallocated memory. This analysis uses the results of the object liveness analysis.

- **Implementation:** We implemented our analyses in the MIT Flex [16] compiler and used them to analyze a set of Java benchmark programs. Our results show that our analyses are able to classify the majority of the allocation sites as unitary allocation sites, and that many such sites can share the same memory. We also implemented and evaluated a compiler optimization that transforms each unitary allocation site to use preallocated memory space instead of invoking the standard memory allocator.

The rest of this chapter is organized as follows. Section 2.2 presents the analysis algorithm. Section 2.3 describes the implementation and presents our experimental results. We discuss related work in Section 2.4 and conclude in Section 2.5.

2.2 Analysis Presentation

Given a program P, the goal of the analysis is to detect pairs of compatible allocation sites from P, i.e., sites that have the property that no object allocated at one site is live at the same time as any object allocated at the other site. Equivalently, the analysis identifies all pairs of *incompatible* allocation sites, i.e., pairs of sites such that an object allocated at the first site and an object allocated at the second site may both be live at the same time in some possible execution of P. An object is live if any of its fields or methods is used in the future. It is easy to prove the following fact:

Fact 1 *Two allocation sites are incompatible if an object allocated at one site is live at the program point that corresponds to the other site.*

To identify the objects that are live at a program point, the analysis needs to track the use of objects throughout the program. There are two complications. First, we have an abstraction problem: the analysis must use a finite abstraction to reason about the potentially unbounded number of objects that the program may create. Second, some parts of the program may read heap references created by other parts of the program. Using a full-fledged, flow-sensitive pointer analysis would substantially increase the time and space requirements of our analysis; a flow-insensitive pointer analysis [185, 21] would not provide sufficient precision since liveness is essentially a flow-sensitive property. We address these complications as follows:

- We use the *object allocation site* model [56]: all objects allocated by a given statement are modelled by an *inside node*[1] associated with that statement's program label.
- The analysis tracks only the objects pointed to by local variables. Nodes whose address may be stored into the heap are said to *escape into the heap.* The analysis conservatively assumes that such a node is not unitary (to ensure this, it sets the node to be incompatible with itself). Notice that, in a usual Java program, there are many objects that are typically manipulated only through local variables: exceptions, iterators, string buffers, etc.[2]

Under these assumptions, a node that does not escape into the heap is live at a given program point if and only if a variable that is live at that program point refers to that node. Variable liveness is a well-studied dataflow analysis [7, 22] and we do not present it here. As a quick reminder, a variable v is live at a program point if and only if there is a path through the control flow graph that starts at that program point, does not contain any definition of v and ends at an instruction that uses v.

The analysis has to process the call instructions accurately. For example, it needs to know the nodes returned from a call and the nodes that escape into the heap

[1]We use the adjective "*inside*" to make the distinction from the "*parameter*" nodes that we introduce later in this chapter.

[2]It is possible to increase the precision of this analyis by tracking one or more levels of heap references (similar to [37]).

n^I_{lb}	$\in$	INode	inside nodes
n^P_k	$\in$	PNode	parameter nodes
n	$\in$	$\mathit{Node} = \mathit{INode} \cup \mathit{PNode}$	general nodes

Figure 2-1: Node Abstraction

during the execution of an invoked method. Reanalyzing each method for each call instruction (which corresponds conceptually to inlining that method) would be inefficient. Instead, we use *parameter nodes* to obtain a single context-sensitive analysis result for each method. The parameter nodes are placeholders for the nodes passed as actual arguments. When the analysis processes a call instruction, it replaces the parameter nodes with the nodes sent as arguments. Hence, the analysis is *compositional*: in the absence of recursion, it analyzes each method exactly once to extract a single analysis result.[3] At each call site, it instantiates the result for the calling context of that particular call site.

Figure 2-1 presents a summary of our node abstraction. We use the following notation: *INode* denotes the set of all inside nodes, *PNode* denotes the set of parameter nodes, and *Node* denotes the set of all nodes. When analyzing a method M, the analysis scope is the method M and all the methods that it transitively invokes. The inside nodes model the objects allocated in this scope. n^I_{lb} denotes the inside node associated with the allocation site from label *lb* (the superscript I stands for "inside"; it is not a free variable). n^I_{lb} represents all objects allocated at label *lb* in the currently analyzed scope. The parameter nodes model the objects that M receives as arguments. The parameter node n^P_i models the object that the currently analyzed method receives as its ith argument of object type.[4]

The analysis has two steps, each one an analysis in itself. The first analysis computes the objects live at each allocation site or call instruction.[5] The second analysis uses the liveness information to compute the incompatibility pairs.

We formulate our analyses as systems of set inclusion constraints and use a bottom-up, iterative fixed-point algorithm to compute the least (under set inclusion) solution of the constraints. For a given program, the number of nodes is bounded by the number of object allocation sites and the number of parameters. Hence, as our constraints are monotonic, all fixed point computations are guaranteed to terminate.

The rest of this section is organized as follows. Section 2.2.1 describes the execution of the analysis on a small example. Section 2.2.2 presents the program representation that the analysis operates on. Section 2.2.3 describes the object liveness analysis. In Section 2.2.4, we describe how to use the object liveness information to compute the incompatibility pairs. Section 2.2.5 discusses how to apply our techniques to multithreaded programs.

[3]The analysis may analyze recursive methods multiple times before it reaches a fixed point.

[4]I.e., not primitive types such as int, char etc.

[5]The object liveness analysis is able to find the live nodes at any program point; however, for efficiency reasons, we produce an analysis result only for the relevant statements.

```
     static void main(String args[]) {
        List l = createList(10);
        filterList(l);
        System.out.println(listToString(l));
     }

     static List createList(int size) {
 1:     List list = new LinkedList();
        for(int i = 0; i < size; i++) {
 2:       Integer v = new Integer(i);
          list.add(v);
        }
        return list;
     }

     static void filterList(List l) {
3a:     for(Iterator it = l.iterator(); it.hasNext();) {
           Integer v = (Integer) it.next();
           if(v.intValue() % 2 == 0)
               it.remove();
        }
     }

     static String listToString(List l) {
 4:     StringBuffer buffer = new StringBuffer();
3b:     for(Iterator it = l.iterator(); it.hasNext();) {
           Integer v = (Integer) it.next();
           buffer.append(v).append(" ");
        }
 5:     return new String(buffer);
     }
```

Figure 2-2: Example Code

2.2.1 Example

Consider the Java code from Figure 2-2. The program creates a linked list that contains the integers from 0 to 9, removes from the list all elements that satisfy a specific condition (the even numbers in our case), then prints a string representation of the remaining list. The program contains six lines that allocate objects. The two `Iterators` from lines 3a and 3b are allocated in library code, at the same allocation site. The other four lines allocate objects directly by executing `new` instructions. For the sake of simplicity, we ignore the other objects allocated in the library. In our example, we have five inside nodes. Node n_1^I represents the linked list allocated at line 1, node n_2^I represents the `Integers` allocated at line 2, etc. The iterators from lines 3a and 3b are both represented by the same node n_3^I (they are allocated at the same site). Figure 2-3 presents the incompatibility graph for this example.

The analysis processes the methods in a bottom-up fashion, starting from the leaves of the call graph. The library method `LinkedList.add` (not shown in Figure 2-2) causes its parameter node n_2^P (n_1^P is the `this` parameter) to escape into the heap (its address is stored in a list cell). `createList` calls `add` with n_2^I as argument; therefore, the analysis instantiates n_2^P with n_2^I and detects that n_2^I escapes. In `filterList`, the parameter node n_1^P (the list) escapes into the heap because `list.iterator()` stores a reference to the underlying list in the iterator that it creates.

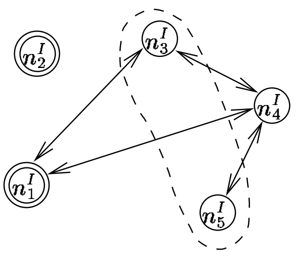


Figure 2-3: Incompatibility graph for the code from Figure 2-2. Circles represent inside nodes; a double circle indicates that the node escapes into the heap. n_3^I and n_5^I are compatible unitary nodes.

In the `listToString` method, n_4^I is live "over the call" to `list.iterator()` that allocates n_3^I: it is pointed to by the local variable `buffer`, which is live both before and after the call. Therefore, n_4^I is incompatible with n_3^I. Because n_4^I is live at line 5, n_4^I is also incompatible with n_5^I. n_3^I is not live at line 5, so n_3^I and n_5^I are still compatible. The parameter node n_1^P (the list) is live at lines 4 and 3b (but not at 5). Therefore, n_1^P is incompatible with n_4^I and n_3^I.

The analysis of `main` detects that `l` points to n_1^I (because `createList` returns n_1^I). As the parameter of `filterList` escapes into the heap, the analysis detects that n_1^I escapes. When processing the call to `listToString`, the analysis instantiates n_1^P with n_1^I and discovers the incompatibility pairs $\langle n_1^I, n_3^I \rangle$ and $\langle n_1^I, n_4^I \rangle$. The analysis has already determined that n_1^I escapes into the heap and is not an unitary node; we generate the last two incompatibility pairs for purely expository purposes.

The graph coloring algorithm colors n_3^I and n_5^I with the same color. This means that the two iterators and the String allocated by the program have the property that no two of them are live at the same time. Hence, the compiler can statically allocate all of these objects into the same memory space.

2.2.2 Program Representation

We work in the context of a *static* compiler that compiles the entire code of the application before the application is deployed and executes. Our compiler provides full reflective access to classes and emulates the dynamic loading of classes precompiled into the executable. It does not support the dynamic loading of classes unknown to the compiler at compile time. This approach is acceptable for our class of target applications, real time software for embedded devices, for which memory consumption analysis is particularly important.

The analyzed program consists of a set of methods $m_1, m_2, \ldots \in \mathit{Method}$, with a distinguished main method. Each method m is represented by its control flow graph CFG_m. The vertices of CFG_m are the labels of the instructions composing m's body, while the edges represent the flow of control inside m. Each method has local variables $v_1, v_2, \ldots v_l \in \mathit{Var}$, and parameters $p_1, \ldots, p_k \in \mathit{Var}$, where Var is the set of local

Name	Format	Informal semantics
COPY	$v_1 = v_2$	copy one local variable into another
NEW	$v = \texttt{new}\ C$	create one object of class C
STORE	$v_1.f = v_2$	create a heap reference
RETURN	$\texttt{return}\ v$	normal return from a method
THROW	$\texttt{throw}\ v$	exceptional return from a method
CALL	$\langle v_N, v_E \rangle = v_1.mn(v_2, \ldots, v_k)$	method invocation
PHI	$v = \phi(v_1, \ldots, v_k)$	SSA ϕ nodes in join points
TYPESWITCH	$\langle v_1, v_2 \rangle = \texttt{typeswitch}\ v : C$	“instanceof” tests

Figure 2-4: Instructions relevant for the analysis.

variables and method parameters.

Figure 2-4 contains the instructions that are relevant for the analysis. We assume that the analyzed program has already been converted into the Single Static Information (SSI) form [17], an extension of the Static Single Assignment (SSA) form [72] (we explain the differences later in this section).

Our intermediate representation models the creation and the propagation of exceptions explicitly. Each instruction that might generate an exception is preceded by a test. If an exceptional situation is detected (e.g., a null pointer dereferencing), our intermediate representation follows the Java convention of allocating and initializing an exception object, (e.g., a `NullPointerException`), then propagating the exception to the appropriate catch block or throwing the exception out of the method if no such block exists. Notice that due to the semantics of the Java programming language, each instruction that can throw an exception is also a potential object allocation site. Moreover, the exception objects are *first class* objects: once an exception is caught, references to it can be stored into the heap or passed as arguments of invoked methods. In practice, we apply an optimization so that each method contains a single allocation site for each automatically inserted exception (for example, NullPointerException and ArrayIndexOutOfBoundsException) that the method may generate but not catch. When the method detects such an exception, it jumps to that allocation site, which allocates the exception object and then executes an exceptional return out of the method.

To allow the inter-procedural propagation of exceptions, a CALL instruction from label lb has two successors: $succ_N(lb)$ for the normal termination of the method and $succ_E(lb)$ for the case when an exception is thrown out of the invoked method.

In both cases — locally generated exceptions or exceptions thrown from an invoked method — the control is passed to the appropriate catch block, if any. This block is determined by a succession of “instanceof” tests. If no applicable block exists, the exception is propagated into the caller of the current method by a THROW instruction “`throw` v”. Unlike a `throw` instruction from the Java language, a THROW instruction from our intermediate representation always terminates the execution of the current method.

Note: we do not check for exceptions that are subclasses of `java.lang.Error`.[6] This is not a significant restriction: as we work in the context of a static compiler, where we know the entire code and class hierarchy, most of these errors cannot be raised by a program that compiled successfully in our system, e.g. `VirtualMachineError`, `NoSuchFieldError` etc. If the program raises any one of the rest of the errors, e.g., `OutOfMemoryError`, it aborts. In most of the cases, this is the intended behavior. In particular, none of our benchmarks catches this kind of exception.

We next present the informal semantics of the instructions from Figure 2-4. A COPY instruction "$v_1 = v_2$" copies the value of local variable v_1 into local variable v_2. A PHI instruction "$v = \phi(v_1, \ldots, v_k)$" is an SSA ϕ node that appears in the join points of the control flow graph; it ensures that each use of a local variable has exactly one reaching definition. If the control arrived in the PHI instruction on the ith incoming edge, v_i is copied into v. A NEW instruction "$v = \texttt{new}\ C$" allocates a new object of class C and stores a reference to it in the local variable v.

A CALL instruction "$\langle v_N, v_E \rangle = v_1.mn(v_2, \ldots, v_k)$" calls the method named mn of the object pointed to by v_1, with the arguments $v_1, \ldots, v_k$.[7] If the execution of the invoked method terminates with a RETURN instruction "$\texttt{return}\ v$", the address of the returned object is stored into v_N and the control flow goes to $succ_N(lb)$, where lb is the label of the call instruction. Otherwise, i.e., if an exception was thrown out of the invoked method, the address of the exception object is stored into v_E and the control flow goes to $succ_E(lb)$.

A TYPESWITCH instruction "$\langle v_1, v_2 \rangle = \texttt{typeswitch}\ v : C$" corresponds to a Java "instanceof" test. It checks whether the class of the object pointed to by v is a subclass of C. v is split into two variables: v_1 is v's restriction on the true branch, while v_2 is v's restriction on the false branch. Therefore, the object pointed to by v_1 is an instance of C, while the object pointed to by v_2 is not. A TYPESWITCH instruction is a simple example of an SSI "sigma" node, "$\langle v_1, v_2 \rangle = \sigma(v)$", that the SSI form introduces to preserve the flow sensitive information acquired in the test instructions. SSI thus allows the elegant construction of predicated dataflow analyses. Apart from this "variable splitting", SSI is similar to the SSA form. In particular, the SSI conversion seems to require linear time in practice [17].

Finally, a STORE instruction "$v_1.f = v_2$" sets the field f of the object referenced by v_1 to point to the object referenced by v_2. The other instructions are irrelevant for our analysis. In particular, as we do not track heap references, the analysis cannot gain any additional information by analyzing the instructions that read references from memory. However, we do analyze the STORE instructions because we need to

[6]In the Java language, these exceptions correspond to severe errors in the virtual machine that the program is not expected to handle.

[7]For the sake of simplicity, in the presentation of the analysis we consider only instance methods (in Java terms, non-static methods), i.e., with v_1 as the `this` argument. The implementation handles both instance methods and static methods.

identify the objects that escape into the heap.

We assume that we have a precomputed call graph: for each label *lb* that corresponds to a CALL instruction, *callees*(*lb*) is the set of methods that that call instruction may invoke. The analysis works with any conservative approximation of the runtime call graph. Our implementation uses a simplified version of the Cartesian Product Algorithm [1].

2.2.3 Object Liveness Analysis

Consider a method M, a label/program point *lb* inside M, and let *live*(*lb*) denote the set of inside and parameter nodes that are live at *lb*. We conservatively consider that a node is live at *lb* iff it is pointed to by one of the variables that are live at that point:

$$\mathit{live}(\mathit{lb}) = \bigcup_{v \text{ live in } \mathit{lb}} P(v)$$

where $P(v)$ is the set of nodes to which v may point. To interpret the results, we need to compute the set E_G of inside nodes that escape into the heap during the execution of the program. To be able to process the calls to M, we also compute the set of nodes that can be normally returned from M, $R_N(M)$, the set of exceptions thrown from M, $R_E(M)$, and the set of parameter nodes that may escape into the heap during the execution of M, $E(M)$. More formally, the analysis computes the following mathematical objects:

$$\begin{array}{rcl} P & : & \mathit{Var} \rightarrow \mathcal{P}(\mathit{Node}) \\ E_G & \subseteq & \mathit{INode} \\ R_N, R_E & : & \mathit{Method} \rightarrow \mathcal{P}(\mathit{Node}) \\ E & : & \mathit{Method} \rightarrow \mathcal{P}(\mathit{PNode}) \end{array}$$

We formulate the analysis as a set inclusion constraint problem. Figure 2-5 presents the constraints generated for a method $M \in \mathit{Method}$ with k parameters $p_1, p_2, \ldots, p_k$. At the beginning of the method, p_i points to the parameter node n_i^P. A COPY instruction "$v_1 = v_2$" sets v_1 to point to all nodes that v_2 points to; accordingly, the analysis generates the constraint $P(v_1) = P(v_2)$.[8] The case of a PHI instruction is similar. A NEW instruction from label *lb*, "$v =$ `new` C", makes v point to the inside node n_{lb}^I attached to that allocation site. The constraints generated for RETURN and THROW add more nodes to $R_N(M)$ and $R_E(M)$, respectively. A STORE instruction "$v_1.f = v_2$", causes all the nodes pointed to by v_2 to escape into the heap. Accordingly, the nodes from $P(v_2)$ are distributed between E_G (the inside nodes) and $E(M)$ (the parameter nodes).

A TYPESWITCH instruction "$\langle v_1, v_2 \rangle =$ `typeswitch` $v : C$" works as a type filter: v_1 points to those nodes from $P(v)$ that may represent objects of a type that is a subtype of C, while v_2 points to those nodes from $P(v)$ that may represent objects

[8] As we use the SSI form, this is the only definition of v_1; therefore, we do not lose any precision by using "=" instead of "$\supseteq$".

Instruction at label lb in method M		Generated constraints
	method entry	$P(p_i) = \{n_i^P\},\ \forall 1 \leq i \leq k$, where $p_1, \ldots, p_k$ are M's parameters.
COPY:	$v_1 = v_2$	$P(v_1) = P(v_2)$
NEW:	$v = \texttt{new}\ C$	$P(v) = \{n_{lb}^I\}$
STORE:	$v_1.f = v_2$	$E(M) \supseteq P(v_2) \cap \mathit{PNode}, \quad E_G \supseteq P(v_2) \cap \mathit{INode}$
RETURN:	$\texttt{return}\ v$	$R_N(M) \supseteq P(v)$
THROW:	$\texttt{throw}\ v$	$R_E(M) \supseteq P(v)$
CALL:	$\langle v_N, v_E\rangle = v_1.mn(v_2, \ldots, v_k)$	$P(v_N) = \bigcup_{m \in callees(lb)} R_N(m)\langle P(v_1), \ldots, P(v_k)\rangle$ $P(v_E) = \bigcup_{m \in callees(lb)} R_E(m)\langle P(v_1), \ldots, P(v_k)\rangle$ $\texttt{let}\ A = \bigcup_{m \in callees(lb)} E(m)\langle P(v_1), \ldots, P(v_k)\rangle\ \texttt{in}$ $E(M) \supseteq A \cap \mathit{PNode}, \quad E_G \supseteq A \cap \mathit{INode}$
PHI:	$v = \phi(v_1, \ldots, v_k)$	$P(v) = \bigcup_{i=1}^k P(v_i)$
TYPESWITCH:	$\langle v_1, v_2\rangle = \texttt{typeswitch}\ v : C$	$P(v_1) = \{n_{lb'}^I \in P(v) \mid type(n_{lb'}^I) \in SubTypes(C)\} \cup \{n^P \in P(v)\}$ $P(v_2) = \{n_{lb'}^I \in P(v) \mid type(n_{lb'}^I) \notin SubTypes(C)\} \cup \{n^P \in P(v)\}$ $SubTypes(C)$ denotes the set of subclasses of class C.

Figure 2-5: Constraints for the object liveness analysis. For each method M, we compute $R_N(M)$, $R_E(M)$, $E(M)$ and $P(v)$ for each variable v live in at a relevant label. We also compute the set E_G of inside nodes that escape into the heap.

of a type that is not a subtype of C. In Figure 2-5, $\mathit{SubTypes}(C)$ denotes the set of all subtypes (i.e., Java subclasses) of C (including C). We can precisely determine the type $\mathit{type}(n^I_{lb'})$ of an inside node $n^I_{lb'}$ by examining the NEW instruction from label lb'. Therefore, we can precisely distribute the inside nodes between $P(v_1)$ and $P(v_2)$. As we do not know the exact types of the objects represented by the parameter nodes, we conservatively put these nodes in both sets.[9]

A CALL instruction "$\langle v_N, v_E\rangle = v_1.mn(v_2,\ \ldots,\ v_k)$" sets v_N to point to the nodes that may be returned from the invoked method(s). For each possible callee $m \in \mathit{callees}(lb)$, we include the nodes from $R_N(m)$ into $P(v_N)$. Note that $R_N(m)$ is a parameterized result. We therefore instantiate $R_N(m)$ before use by replacing each parameter node n^P_i with the nodes that the corresponding argument v_i points to, i.e., the nodes from $P(v_i)$. The case of v_E is analogous. The execution of the invoked method m may also cause some of the nodes passed as arguments to escape into the heap. Accordingly, the analysis generates a constraint that instantiates the set $E(m)$ and the uses the nodes from the resulting set $E(m)\langle P(v_1), \ldots, P(v_k)\rangle$ to update E_G and $E(M)$.

Here is a more formal and general definition of the previously mentioned instantiation operation: if $S \subseteq \mathit{Node}$ is a set that contains some of the parameter nodes $n^P_1, \ldots, n^P_k$ (not necessarily all), and $S_1, \ldots, S_k \subseteq \mathit{Node}$, then

$$S\langle S_1, \ldots S_k\rangle \ = \{n^I \in S\} \ \cup \ \textstyle\bigcup_{n^P_i \in S} S_i$$

2.2.4 Computing the Incompatibility Pairs

Once the computation of the object liveness information completes, the analysis computes the (global) set of pairs of incompatible allocation sites $\mathit{Inc}_G \subseteq \mathit{INode} \times \mathit{INode}$.[10] The analysis uses this set of incompatible allocation sites to detect the unitary allocation sites and to construct the compatibility classes.

Figure 2-6 presents the constraints used to compute Inc_G. An allocation site from label lb is incompatible with all the allocation sites whose corresponding nodes are live at lb.

However, as some of the nodes from $\mathit{live}(lb)$ may be parameter nodes, we cannot generate all incompatibility pairs directly. Instead, for each method M, the analysis collects the incompatibility pairs involving one parameter node into a set of *parametric* incompatibilities $\mathit{ParInc}(M)$. It instantiates this set at each call to M, similar to the way it instantiates $R_N(M)$, $R_E(M)$ and $E(M)$:

$$\mathit{ParInc}(M)\langle S_1, \ldots, S_k\rangle \ = \textstyle\bigcup_{\langle n^P_i, n\rangle \in \mathit{ParInc}(M)} S_i \times \{n\}$$

(S_i is the set of nodes that the ith argument sent to M might point to). Notice that some S_i may contain a parameter node from M's caller. However, at some point in

[9]A better solution would be to consider the declared type C_p of the corresponding parameter and check that C_p and C have at least one common subtype.

[10]Recall that there is a bijection between the inside nodes and the allocation sites.

Instruction at label lb in method M	Generated constraints
$v = \texttt{new}\ C$	$live(lb) \times \{n^I_{lb}\} \subseteq AllInc(M)$
$\langle v_N, v_E\rangle = v_1.mn(v_2, \ldots, v_k)$ $\swarrow \qquad \searrow$ $succ_N(lb) \qquad succ_E(lb)$	$\forall m \in callees(lb)$, $ParInc(m)\langle P(v_1), \ldots, P(v_k)\rangle \subseteq AllInc(M)$ $(live(lb) \cap live(succ_N(lb))) \times A_N(m) \subseteq AllInc(M)$ $(live(lb) \cap live(succ_E(lb))) \times A_E(m) \subseteq AllInc(M)$
$\forall M \in Method$, $AllInc(M) \cap (INode \times INode) \subseteq Inc_G$ $AllInc(M) \setminus (INode \times INode) \subseteq ParInc(M)$	

Figure 2-6: Constraints for computing the set of incompatibility pairs.

Instruction at label lb in method M	Condition	Generated constraints
$v = \texttt{new}\ C$	$lb \leadsto \texttt{return}$	$n^I_{lb} \in A_N(M)$
	$lb \leadsto \texttt{throw}$	$n^I_{lb} \in A_E(M)$
$\langle v_N, v_E\rangle = v_1.mn(v_2, \ldots, v_k)$	$succ_N(lb) \leadsto \texttt{return}$	$A_N(m) \subseteq A_N(M), \forall m \in callees(lb)$
	$succ_N(lb) \leadsto \texttt{throw}$	$A_N(m) \subseteq A_E(M), \forall m \in callees(lb)$
	$succ_E(lb) \leadsto \texttt{return}$	$A_E(m) \subseteq A_N(M), \forall m \in callees(lb)$
	$succ_E(lb) \leadsto \texttt{throw}$	$A_E(m) \subseteq A_E(M), \forall m \in callees(lb)$

Figure 2-7: Constraints for computing A_N, A_E. For each relevant instruction, if the condition from the second column is satisfied, the corresponding constraint from the third column is generated.

the call graph, each incompatibility pair will involve only inside nodes and will be passed to Inc_G.

To simplify the equations from Figure 2-6, for each method M, we compute the entire set of incompatibility pairs $AllInc(M)$. After $AllInc(M)$ is computed, the pairs that contain only inside nodes are put in the global set of incompatibilities Inc_G; the pair that contains a parameter node are put in $ParInc(M)$. Our implementation of this algorithm performs this separation "on the fly", as soon as an incompatibility pair is generated, without the need for $AllInc(M)$.

In the case of a CALL instruction, we have two kinds of incompatibility pairs. We have already mentioned the first kind: the pairs obtained by instantiating $ParInc(m), \forall m \in callees(lb)$. In addition, each node that is live "over the call" (i.e., before and after the call) is incompatible with all the nodes corresponding to the allocation sites from the invoked methods. To increase the precision, we treat the normal and the exceptional exit from an invoked method separately. Let $A_N(m) \subseteq INode$ be the set of inside nodes that represent the objects that may be allocated during a method execution that returns normally. Similarly, let $A_E(m) \subseteq INode$ be the set of inside nodes that represent the objects that may be allocated during an invocation of m that returns with an exception. We describe later how to compute these sets; for the moment we suppose the analysis computes them just before it starts to generate the incompatibility pairs. Let $\mathit{succ}_N(\mathit{lb})$ be the successor corresponding to the normal return from the CALL instruction from label lb. The nodes from $\mathit{live}(\mathit{lb}) \cap \mathit{live}(\mathit{succ}_N(\mathit{lb}))$ are incompatible with all nodes from $A_N(m)$. A similar relation holds for $A_E(m)$.

Computation of $A_N(M)$, $A_E(M)$

Given a label lb from the code of some method M, we define the predicate "$\mathit{lb} \rightsquigarrow \texttt{return}$" to be true iff there is a path in CFG_M from lb to a RETURN instruction (i.e., the instruction from label lb may be executed in an invocation of M that returns normally). Analogously, we define "$\mathit{lb} \rightsquigarrow \texttt{throw}$" to be true iff there is a path from lb to a THROW instruction. Computing these predicates is an easy graph reachability problem. For a method M, $A_N(M)$ contains each inside node n^I_{lb} that corresponds to a NEW instruction at label lb such that $\mathit{lb} \rightsquigarrow \texttt{return}$. In addition, for a CALL instruction from label lb in M's code, if $\mathit{succ}_N(\mathit{lb}) \rightsquigarrow \texttt{return}$, then we add all nodes from $A_N(m)$ into $A_N(M)$, for each possible callee m. Analogously, if $\mathit{succ}_E(\mathit{lb}) \rightsquigarrow \texttt{return}$, $A_E(m) \subseteq A_N(M)$. The computation of $A_E(m)$ is similar. Figure 2-7 formally presents the constraints for computing the sets $A_N(M)$ and $A_E(M)$.

2.2.5 Multithreaded Applications

So far, we have presented the analysis in the context of a single-threaded application. For a multithreaded application, the analysis needs to examine all methods that are transitively called from the main method and from the `run()` methods of the threads that may be started. In addition, all nodes that correspond to started threads need to be marked as escaped nodes. The rest of the analysis is unchanged.

In Java, each thread is represented by a thread object allocated in the heap. For an object to escape one thread to be accessed by another, it must be reachable from either the thread object or a static class variable (global variables are called static class variables in Java). In both cases, the analysis determines that the corresponding allocation site is not unitary. Therefore, all objects allocated at unitary allocation sites are local to the thread that created them and do not escape to other threads. Although we know that no two objects allocated by the same thread at the same unitary site are live at any given moment, we can have multiple live objects allocated at this site by different threads. Hence, for each group of compatible unitary sites, we need to allocate one memory slot *per thread*, instead of one per program.

The compiler generates code such that each time the program starts a new thread, it preallocates memory space for all unitary allocation sites that may be executed by that thread. For each unitary allocation site, the compiler generates code that retrieves the current thread and uses the preallocated memory space for the unitary site in the current thread. When a thread terminates its execution, it deallocates its preallocated memory space. As only thread-local objects used that space, this deallocation does not create dangling references. To bound the memory space occupied by the unitary allocation sites, we need to bound the number of threads that simultaneously execute in the program at any given time.

2.2.6 Optimization for Single-Thread Programs

In the previous sections, we consider a node that escapes into the heap to be incompatible with all other nodes, including itself. This is equivalent to considering the node to be live during the *entire* program. We can gain additional precision by considering that once a node escapes, it is live only for *the rest* of the program. This enhancement allows us to preallocate even objects that escape into the heap, if their allocation site executes at most once. This section presents the changes to our analysis that apply this idea.

We no longer use the global set E_G. Instead, for each label *lb*, $E(lb) \subseteq Node$ denotes the set of nodes that the instruction at label *lb* may store a reference to into the heap. This set is relevant only for labels that correspond to STOREs and CALLs; for a CALL, it represents the nodes that escape during the execution of the invoked method.

We extend the set of objects live at label *lb* (from method M) to include all objects that are escaped by instructions at labels *lb'* from M that can reach *lb* in CFG$_M$:

$$live(lb) = \bigcup_{v \text{ live in } lb} P(v) \;\cup\; \bigcup_{\substack{lb' \text{ in } M \\ lb' \rightsquigarrow lb}} E(lb')$$

We change the constraints from Figure 2-5 as follows: for a STORE instruction "$v_1.f = v_2$", we generate only the constraint $E(lb) = P(v_2)$. For a CALL instruction "$\langle v_N, v_E \rangle = v_1.mn(v_2, \ldots, v_k)$", we generate the same constraints as before for $P(v_N)$ and $P(v_E)$, and the additional constraint

$$E(lb) = \bigcup_{m \in callees(lb)} E(m)\langle P(v_1), \ldots, P(v_k)\rangle$$

The rules for STORE and CALL no longer generate any constraints for E_G (unused now) and $E(M)$. Instead, we define $E(M)$ as

$$E(M) = \bigcup_{lb \text{ in } M} E(lb)$$

Now, $E(M) \subseteq \mathcal{P}(Node)$ denotes the set of all nodes — not only parameter nodes as before, but also inside nodes — that escape into the heap during M's execution.

The rest of the analysis is unchanged. The new definition of *live*(lb) ensures that if a node escapes into the heap at some program point, it is incompatible with all nodes that are live at any future program point. Notice that objects allocated at unitary sites are no longer guaranteed to be thread local, and we cannot apply the preallocation optimization described at the end of Section 2.2.5. Therefore, we use this version of the analysis only for single thread programs.

Application	Description
SPECjvm98 benchmark set	
_200_check	Simple program; tests JVM features
_201_compress	File compression tool
_202_jess	Expert system shell
_209_db	Database application
_213_javac	JDK 1.0.2 Java compiler
_222_mpegaudio	Audio file decompression tool
_228_jack	Java parser generator
Java Olden benchmark set	
BH	Barnes-Hut N-body solver
BiSort	Bitonic Sort
Em3d	Models the propagation of electromagnetic waves through 3D objects
Health	Simulates a health-care system
MST	Computes the minimum spanning tree in a graph using Bentley's algorithm
Perimeter	Computes the perimeter of a region in a binary image represented by a quadtree
Power	Maximizes the economic efficiency of a community of power consumers
TSP	Solves the traveling salesman problem using a randomized algorithm
TreeAdd	Recursive depth-first traversal of a tree to sum the node values
Voronoi	Computes a Voronoi diagram for a random set of points
Miscellaneous	
_205_raytrace	Single thread raytracer (not an official part of SPECjvm98)
JLex	Java lexer generator
JavaCUP	Java parser generator

Table 2.1: Analyzed Applications

2.3 Experimental Results

We have implemented our analysis, including the optimization from Section 2.2.6, in the MIT Flex compiler system [16]. We have also implemented the compiler transformation for memory preallocation: our compiler generates executables with the property that unitary sites use preallocated memory space instead of calling the memory allocation primitive. The memory for these sites is preallocated at the beginning of the program. Our implementation does not currently support multithreaded programs as described in Section 2.2.5.

We measure the effectiveness of our analysis by using it to find unitary allocation sites in a set of Java programs. We obtained our results on a Pentium 4 2.8Ghz system with 2GB of memory running RedHat Linux 7.3. We ran our compiler and analysis using Sun JDK 1.4.1 (hotspot, mixed mode); the compiler generates native executables that we ran on the same machine. Table 2.1 presents a description of the programs in our benchmark suite. We analyze programs from the SPECjvm98 benchmark suite[11] and from the Java version of the Olden benchmark suite [52, 51]. In addition, we analyze JLex, JavaCUP, and _205_raytrace.

Table 2.2 presents several statistics that indicate the size of each benchmark and the analysis time. The statistics refer to the user code plus all library methods called from the user code. As the data in Table 2.2 indicate, in general, the time required to perform our analysis is of the same order of magnitude as the time required to build

[11]With the exception of _227_mtrt, which is multithreaded.

Application	Analyzed methods	Bytecode instrs	SSI IR size (instr.)	SSI conversion time (s)	Analysis time (s)
_200_check	208	7962	10353	1.1	4.1
_201_compress	314	8343	11869	1.2	7.4
_202_jess	1048	31061	44746	5.3	101.2
_209_db	394	12878	18162	2.7	12.3
_213_javac	1681	52941	71050	8.2	1126.2
_222_mpegaudio	511	18041	30884	5.2	15.9
_228_jack	618	23864	37253	11.6	55.6
BH	169	6476	8690	1.4	3.6
BiSort	123	5157	6615	1.2	2.9
Em3d	142	5519	7497	0.9	3.1
Health	141	5803	7561	0.9	3.2
MST	139	5228	6874	1.2	3.0
Perimeter	144	5401	6904	1.2	2.7
Power	135	6039	7928	1.0	3.2
TSP	127	5601	6904	0.9	3.1
TreeAdd	112	4814	6240	0.8	2.8
Voronoi	274	8072	10969	1.8	4.3
_205_raytrace	498	14116	20875	4.2	23.0
JLex	482	22306	31354	4.0	12.3
JavaCUP	769	27977	41308	5.8	32.0

Table 2.2: Analyzed Code Size and Analysis Time

the intermediate representation of the program. The only exceptions are _202_jess and _213_javac.

Table 2.3 presents the number of total allocation sites and unitary allocation sites in each program. These results show that our analysis is usually able to identify the majority of these sites as unitary sites: of the 14065 allocation sites in our benchmarks, our analysis is able to classify 8396 (60%) as unitary sites. For twelve of our twenty benchmarks, the analysis is able to recognize over 80% of the allocation sites as unitary.

Table 2.3 also presents results for the allocation sites that allocate exceptions (i.e., any subclass of java.lang.Throwable), non-exceptions (the rest of the objects), and java.lang.StringBuffers (a special case of non-exceptions). For each category, we present the total number of allocation sites of that kind and the proportion of these sites that are unitary. The majority of the unitary allocation sites in our benchmarks allocate exception or string buffer objects. Of the 9660 total exception allocation sites in our benchmarks, our analysis is able to recognize 6602 (68%) as unitary sites. For thirteen of our twenty benchmarks, the analysis is able to recognize over 90% of the exception allocation sites as unitary sites. Of the 1293 string buffer allocation sites, our analysis is able to recognize 1190 (92%) as unitary sites. For eight benchmarks, the analysis is able to recognize over 95% of the string buffer allocation sites as unitary sites.

Table 2.4 presents the size of the statically preallocated memory area that is used to store the objects created at unitary allocation sites. The second column of the table presents results for the case where each unitary allocation site has its own preallocated memory chunk. As described in the chapter introduction, we can decrease the

Application	Allocation sites	Unitary sites count	Unitary sites %	Exceptions total	Exceptions unitary %	Non-exceptions total	Non-exceptions unitary %	StringBuffers total	StringBuffers unitary %
_200_check	407	326	80%	273	92%	134	57%	44	97%
_201_compress	489	155	32%	390	28%	99	44%	38	97%
_202_jess	1823	919	50%	1130	58%	693	38%	233	84%
_209_db	736	354	48%	565	48%	171	49%	65	98%
_213_javac	2827	1086	38%	1863	47%	964	23%	195	89%
_222_mpegaudio	825	390	47%	625	55%	200	24%	43	97%
_228_jack	910	479	53%	612	54%	298	50%	135	99%
BH	329	281	85%	243	98%	86	51%	18	94%
BiSort	234	198	85%	177	97%	57	47%	17	94%
Em3d	276	235	85%	206	98%	70	50%	20	95%
Health	276	227	82%	202	97%	74	42%	17	94%
MST	257	216	85%	194	97%	63	44%	16	93%
Perimeter	239	200	84%	180	97%	59	45%	16	93%
Power	262	213	81%	192	97%	70	39%	15	93%
TSP	235	199	85%	176	97%	59	49%	17	94%
TreeAdd	227	190	84%	170	96%	57	46%	15	93%
Voronoi	448	387	86%	349	98%	99	44%	28	96%
_205_raytrace	753	318	42%	525	44%	228	39%	43	95%
JLex	971	812	84%	645	99%	326	54%	72	86%
JavaCUP	1541	1211	79%	943	93%	598	56%	246	92%
Total	14065	8396	60%	9660	68%	4405	41%	1293	92%

Table 2.3: Unitary Site Analysis Results

Application	Preallocated memory size (bytes) normal	Preallocated memory size (bytes) sharing	Size reduction %
_200_check	5516	196	96%
_201_compress	2676	144	95%
_202_jess	17000	840	96%
_209_db	6028	252	96%
_213_javac	18316	332	98%
_222_mpegaudio	6452	104	98%
_228_jack	8344	224	97%
BH	4604	224	95%
BiSort	3252	96	98%
Em3d	3860	200	95%
Health	3716	96	97%
MST	3532	96	97%
Perimeter	3280	96	98%
Power	3540	196	94%
TSP	3292	104	97%
TreeAdd	3120	92	98%
Voronoi	6368	192	97%
_205_raytrace	5656	644	89%
JLex	13996	1676	88%
JavaCUP	20540	1180	94%
Total	143088	6984	95%

Table 2.4: Preallocated Memory Size

Application	Total objects	Preallocated objects	
		count	%
_200_check	725	238	33%
_201_compress	941	108	11%
_202_jess	7917932	3275	0%
_209_db	3203535	142	0%
_213_javac	5763881	335775	6%
_222_mpegaudio	1189	7	1%
_228_jack	6857090	409939	6%
BH	15115028	7257600	48%
BiSort	131128	15	0%
Em3d	16061	23	0%
Health	1196846	681872	57%
MST	2099256	1038	0%
Perimeter	452953	10	0%
Power	783439	12	0%
TSP	49193	32778	67%
TreeAdd	1048620	13	0%
Voronoi	1431967	16399	1%
_205_raytrace	6350085	4080258	64%
JLex	1419852	12926	1%
JavaCUP	100026	16517	17%

Table 2.5: Preallocated Objects

preallocated memory size significantly if we use a graph coloring algorithm to allow compatible unitary allocation sites to share the same preallocated memory area. The third column of Table 2.4 presents results for this case. Our compiler optimization always uses the graph coloring algorithm; we provide the second column for comparison purposes only. The graph coloring algorithm finds an approximation of the smallest number of colors such that no two incompatible allocation sites have the same color. For each color, we preallocate a memory area whose size is the maximum size of the classes allocated at allocation sites with that color. Our implementation uses the DSATUR graph coloring heuristic [49]. It is important to notice that the DSATUR heuristic minimizes the numbers of colors, not the final total size of the preallocated memory. However, this does not appear to have a significant negative effect on our results: as the numbers from Table 2.4 show, we are able to reduce the preallocated memory size by at least 88% in all cases; the average reduction is 95%.

Theoretically, the preallocation optimization may allocate more memory than the original program: preallocating a memory area for a set of compatible allocation sites reserves that area for the entire lifetime of the program, even when no object allocated at the attached set of compatible sites is reachable. An extreme case is represented by the memory areas that we preallocate for allocation sites that the program never executes. However, as the data from Table 2.4 indicate, in practice, the amount of preallocated memory for each analyzed application is quite small.

We compiled each benchmark with the memory preallocation optimization enabled. Each optimized executable finished normally and produced the same result as the unoptimized version. We executed the SPECjvm98 and the Olden applications with their default workload. We ran JLex and JavaCUP on the lexer and parser files from our compiler infrastructure. We instrumented the allocation sites to measure how many objects were allocated by the program and how many of these objects used the preallocated memory. Table 2.5 presents the results of our measurements. For five of our benchmarks, at least one third of the objects resided in the preallocated memory. There is no correlation between the static number of unitary sites and the dynamic number of objects allocated at those sites. This is explained by the large difference in the number of times different allocation sites are executed. In general, application-specific details tend to be the only factor in determining these dynamic numbers. For example, in JLex, 95% of the objects are iterators allocated at the same (non-unitary) allocation site; _213_javac and JavaCUP use many StringBuffers that we can preallocate; both _205_raytrace and BH use many temporary objects to represent mathematical vectors, etc.

2.4 Related Work

To the best of our knowledge, we present the first use of a pointer analysis to enable static object preallocation. Other researchers have used pointer and/or escape analyses to improve the memory management of Java programs [58, 202, 35], but these algorithms focus on allocating objects on the call stack. Researchers have also developed algorithms that correlate the lifetimes of objects with the lifetimes of invoked methods, then use this information to allocate objects in different regions [194]. The goal is to eliminate garbage collection overhead by atomically deallocating all of the objects allocated in a given region when the corresponding function returns. Other researchers [111] require the programmer to provide annotations (via a rich type systems) that specify the region that each object is allocated into.

Bogda and Hoelzle [37] use pointer analysis to eliminate unnecessary synchronizations in Java programs. In spite of the different goals, their pointer analysis has many technical similarities with our analysis. Both analyses avoid maintaining precise information about objects that are placed "too deep" into the heap. Bogda and Hoelzle's analysis is more precise in that it can stack allocate objects reachable from a single level of heap references, while our analysis does not attempt to maintain precise points-to information for objects reachable from the heap. On the other hand, our analysis is more precise in that it computes live ranges of objects and treats exceptions with more precision. In particular, we found that our predicated analysis of type switches (which takes the type of the referenced object into account) was necessary to give our analysis enough precision to statically preallocate exception objects.

Our analysis has more aggressive aims than escape analysis. Escape analysis is typically used to infer that the lifetimes of all objects allocated at a specific allocation site are contained within the lifetime of either the method that allocates them or one of the methods that (transitively) invokes the allocating method. The compiler can transform such an allocation site to allocate the object from the method stack frame instead of the heap. Notice that the analysis does not provide any bound on the number of objects allocated at that allocation site: in the presence of recursion or loops, there may be an arbitrary number of live objects from a single allocation site (and an arbitrary number of these objects allocated on the call stack). In contrast, our analysis identify allocation sites that have the property that at most one object is live at any given time.

In addition, the stack allocation transformation may require the compiler to lift the corresponding object allocation site out of the method that originally contained it to one of the (transitive) callers of this original allocating method [202]. The object would then be passed by reference down the call stack, incurring runtime overhead.[12] The static preallocation optimization enabled by our analysis does not suffer from this drawback. The compiler transforms the original allocation site to simply acquire a pointer to the statically allocated memory; there is no need to move the allocation site into the callers of the original allocating method.

Our combined liveness and incompatibility analysis and use of graph coloring to minimize the amount of memory required to store objects allocated at unitary allocation sites is similar in spirit to register allocation algorithms [22, Chapter 11]. However, register allocation algorithms are concerned only with the liveness of the local variables, which can be computed by a simple intraprocedural analysis. We found that obtaining useful liveness results for dynamically allocated objects is significantly more difficult. In particular, we found that we had to use a predicated analysis and track the flow of objects across procedure boundaries to identify significant amounts of unitary sites.

2.5 Conclusions

We have presented an analysis designed to simplify the computation of an accurate upper bound on the amount of memory required to execute a program. This analysis statically preallocates memory to store objects allocated at unitary allocation sites and enables objects allocated at compatible unitary allocation sites to share the same preallocated memory. Our experimental results show that, for our set of Java benchmark programs, 60% of the allocation sites are unitary and can be statically preallocated. Moreover, allowing compatible unitary allocation sites to share the same preallocated memory leads to a 95% reduction in the amount of memory required for these sites. Based on this set of results, we believe our analysis can automatically and effectively eliminate the need to consider many object allocation sites when computing an accurate upper bound on the amount of memory required to execute the program. We have also used the analysis to optimize the memory managment.

[12]A semantically equivalent alternative is to perform method inlining. However, inlining introduces its own set of overheads.

Chapter 3

Data Size Optimizations for Java Programs

3.1 Introduction

We present a set of techniques for reducing the amount of data space required to represent objects in object-oriented programs. Our techniques optimize the representation of both the programmer-defined fields within each object and the header information used by the run-time system:

- **Field Reduction:** Our flow-sensitive, interprocedural bitwidth analysis computes the range of values that the program may assign to each field. The compiler then transforms the program to reduce the size of the field to the smallest type capable of storing that range of values.

- **Unread and Constant Field Elimination:** If the bitwidth analysis finds that a field always holds the same constant value, the compiler eliminates the field. It removes each write to the field, and replaces each read with the constant value. Fields without executable reads are also removed.

- **Static Specialization:** Our analysis finds classes with fields whose values do not change after initialization, even though different instances of the object may have different values for these fields. It then generates specialized versions of each class which omit these fields, substituting accessor methods which return constant values.

- **Field Externalization:** Our analysis uses profiling to find fields that almost always have the same default value. It then removes these fields from their enclosing class, using a hash table to store only values of the field that differ from the default value. It replaces writes to the field with an insertion into the hash table (if the written value is not the default value) or a removal from the hash table (if the written value is the default value). It replaces reads with hash table lookups; if the object is not present in the hash table, the lookup simply returns the default value.

- **Class Pointer Compression:** We use rapid type analysis to compute an upper bound on the number of classes that the program may instantiate. Objects in standard Java implementations have a header field, commonly called `claz`, which contains a pointer to the class data for that object, such as inheritance information and method dispatch tables. Our compiler uses the results of the analysis to replace the reference with a smaller offset into a table of pointers to the class data.

- **Byte Packing:** All of the above transformations may reduce or eliminate the amount of space required to store each field in the object or object header. Our byte packing algorithm arranges the fields in the object to minimize the object size.

All of these transformations reduce the space required to store objects, but some potentially increase the running time of the program. Our experimental results show that, for our set of benchmark programs, all of our techniques combined can reduce the peak amount of memory required to run the program by as much as 40%, although the running time may increase. In a memory-limited embedded system where performance is not critical, cost savings may directly result from the reduced minimum heap size.

3.1.1 Contributions

This paper makes the following contributions:

- **Space Reduction Transformations:** It presents a set of novel transformations for reducing the memory required to represent objects in object-oriented programs.

- **Analysis Algorithms:** It presents a set of analysis algorithms that automatically extract the information required to apply the space reduction transformations.

- **Implementation:** We have fully implemented all of the analyses and techniques presented in the paper. Our experience with this implementation enables us to discuss the pragmatic details necessary for an effective implementation of our techniques.

- **Experimental Results:** This paper presents a set of experimental results that characterize the impact of our transformations, revealing the extent of the savings available and the performance cost of attaining them.

3.2 Examples

We next present a pair of examples that illustrate the kinds of analyses and transformations that our compiler performs.

```
public class JValue {
   int integerType = 0;
   int floatType = 1;
   int type, positive;
   Object value;
   void setInteger(Integer i) {
      type = integerType; value = i;
      positive = (i.intValue() > 0) ?  1 :  0;
   }
   void setFloat(Float f) {
      type = floatType; value = f;
      positive = (f.floatValue() > 0) ?  1 :  0;
   }
}
```

Figure 3-1: The `JValue` class.

3.2.1 Field Reduction and Constant Field Elimination

Figure 3-1 presents the `JValue` class, which is a wrapper around either an `Integer` object or a `Float` object. The `type` field indicates which kind of object is stored in the `value` field of the class, essentially implementing a tagged union.[1] The class also maintains the `positive` field, which is `1` if the wrapped number is positive and `0` otherwise.

Our bitwidth analysis uses an interprocedural value-flow algorithm to compute upper and lower bounds for the values that can appear in each variable. This analysis tracks the flow of values across procedure boundaries via parameters, into and out of the heap via instance variables of classes, and through intermediate temporaries and local variables in the program. It also reasons about the semantics of arithmetic operators such as `+` and `*` to obtain bounds for the values computed by arithmetic expressions. Assume that the analysis examines the rest of the program (not shown) and discovers the following facts about how the program uses this class: a) the `integerType` field always has the value `0`, b) the `floatType` field always has the value `1`, c) the `type` field always has a value between `0` and `1` (inclusive), and d) the `positive` field always has a value between `0` and `1` (also inclusive).

Our compiler uses this information to remove all occurrences of the `integerType` and `floatType` fields from the program. It replaces each read of the `integerType` field with the constant `0`, and each read of the `floatType` field with the constant `1`. It also uses the bounds on the values of the `type` and `positive` variables to reduce the size of the corresponding fields. Our currently implemented compiler rounds field sizes to the nearest byte required to hold the range of values that can occur. Our byte packing algorithm then generates a dense packing of the values, attempting to preserve the alignment of the variables if possible. In this case, the algorithm can reduce the field sizes by six bytes and the overall size of the object by one four-byte word. If the runtime can support unaligned objects without external fragmentation,

[1]This class is a simplified version of similar classes that appear in some of our benchmarks. See for example the `jess.Value` class in SPECjvm98 benchmark `jess`.

```
public final class String {
   private final char value[];
   private final int offset;
   private final int count;
   ...
   public char charAt(int i) {
      return value[offset+i];
   }
   public String substring(int start)
   {
      int noff = offset + start;
      int ncnt = count - start;
      return new String(noff, ncnt, value);
   }
}
```

Figure 3-2: Portions of the `java.lang.String` class.

we can reduce the size of all allocated `JValue` objects by the full six bytes.

3.2.2 Static Specialization

Figure 3-2 presents portions of the implementation of the `java.lang.String` class from the Java standard class library. The `value` field in this class refers to a character array that holds the characters in the string; the `count` field holds the length of the string. In some cases, instances of the `String` class are derived substrings of other instances (see the `substring` method in Figure 3-2), in which case the `offset` field provides the offset of the starting point of the string within a shared `value` character array. Note that the `value`, `offset`, and `count` fields are all initialized when the string is constructed and do not change during the lifetime of the string.

In practice, most strings are not created as explicit substrings of other strings, so the `offset` field in most strings is zero. In fact, all of the public `String` constructors create strings with `offset` zero; only the `substring` method creates strings with a nonzero offset. And even at calls to the private `String(int, int, char[])` constructor inside the `substring` method, it is possible to dynamically test the values of the parameters at the allocation site to determine if the newly constructed string will have a zero or nonzero offset.

Our analysis exploits this fact by splitting the `String` class into two classes: a superclass `SmallString` that omits the `offset` field, and a subclass `BigString` that extends `SmallString` and includes the `offset` field. Each of these two new classes implements a `getOffset()` method to replace the field: the `getOffset()` method in the `SmallString` class simply returns zero; but the `getOffset()` method in the `BigString` class returns the value of the `offset` field in `BigString`. Figure 3-3 illustrates this transformation.

At every allocation site except the one inside the `substring` method, the transformed program allocates a `SmallString` object. Inside the `substring` method, the program generates code that dynamically tests if the offset in the substring will be zero. If so, it allocates a `SmallString` object; if not, it allocates a `BigString` object. (See Figure 3-4.) This transformation therefore eliminates the `offset` field in the

```
public final class SmallString {
   private final char value[];
   private final int count;
   int getOffset() { return 0; }
   ...
   public char charAt(int i) {
      return value[getOffset()+i];
   }
}
public final class BigString extends SmallString {
   private final int offset;
   int getOffset() { return offset; }
}
```

Figure 3-3: Static specialization of `java.lang.String`.

majority of strings.

The analysis required to support this transformation takes place in two phases. The first phase scans the program to identify fields that are amenable to transformation.[2] In our example, the analysis determines that the `offset` field is never written after it is initialized. In the next phase, we determine if the initialized value of the field can be determined before the object is created, by examining the specific constructor invoked and its parameters. In our example, the analysis determines that the `offset` field is zero for all constructors except the private constructor invoked within the `substring` method. It also determines that, for objects created within substring, the value of the `offset` field is simply the value of the `noff` parameter to this constructor.

This analysis identifies a set of candidate fields. The analysis chooses one of the candidate fields, then splits the class along the possible values that can appear in the field. Our current implementation uses profiling to select the field that will provide the largest space savings; our policy takes both the size of the field and the percentage of objects that have the same value for that field. In our example, the analysis identifies the `offset` field as the best candidate and splits the class on that field. We can apply this idea recursively to the new program to obtain the benefits of splitting on multiple fields.

In this example all of the relevant fields are `private`, which would, in principle, enable an implementation to apply the optimization with an analysis of only the `String` class. Our analysis, however, is powerful enough to examine the rest of the program and discover the facts required to apply the optimization in the absence of `private` or `final` declarations and even for fields accessed outside their declaring class.

3.2.3 Field Externalization

In the string example discussed above, it was possible to determine which version of the specialized class to use at object allocation time. In some cases, however, a

[2]See Section 3.3.5 for a precise definition.

```
public SmallString substring(int start)
{
   int noff = offset + start;
   int ncnt = count - start;
   if (noff==0)
      return new SmallString(value, noff, ncnt);
   else
      return new BigString(value, noff, ncnt);
}
```

Figure 3-4: Dynamic selection among specialized classes in a method from `java.lang.String`.

given field may almost always have a given value, even though it is not possible to statically determine when the value might be changed or which objects will contain fields of that value. In such cases we apply another optimization, *field externalization*. This optimization removes the field from the class, replacing fields whose values differ from the default value with hash table entries that map objects to values. If an object/value mapping is present in the hash table, that entry provides the value of the removed field. If there is no mapping for a given object, the field is assumed to have the default value. In our current implementation, we use profiling to identify the default value.

In this scheme, writes to the field are converted into a check to see if the new value of the field is the default value. If so, the generated code simply removes any old mappings for that object from the hash table. If not, the generated code replaces any old mapping with a new mapping recording the new value.

3.2.4 Hash/Lock Externalization

Our currently implemented system applies field externalization in a general way to any field in the object. We would, however, like to highlight an especially useful extension of the basic technique. Java implementations typically store an object hash code and lock information in the object header. For many objects, however, the program never actually uses the hash code or lock information. Our implemented system therefore uses a variant of field externalization called *hash/lock externalization*. This variant allocates all objects without the hash code and lock information fields in the header, then lazily creates the fields when necessary. Specifically, if the program ever uses the hash code or lock information, the generated code creates the hash code or lock information for the object, then stores this information in a table mapping objects to their hash code or lock information.[3]

Note that, in general, this transformation (as well as field externalization) may actually increase space usage. But in practice, we have found that our set of benchmark programs rarely uses these fields. The overall result is a substantial space savings. The combination of class pointer compression and hash/lock elimination can produce a common-case object header size of one byte—one byte for a class index and no

[3]The object's address is used as its key when field externalization is done. The garbage collector is responsible for updating the field entries if it moves objects, by rehashing on the new address.

space at all for hash code or lock.

3.3 Analysis Algorithms

In this section we will present details of the analyses that enable our transformations.

3.3.1 Rapid Type Analysis

We start with a rapid type analysis [26] to collect the set of instantiated classes and callable methods. This analysis allows us to generate a conservative call graph for the program, using the known receiver type at the call-site and its set of instantiated subclasses in the hierarchy. Based on the class hierarchy, we can also tag all leaf classes as `final`, regardless of whether the source code contained this modifier. Methods which are not overridden, based on the hierarchy, are also marked `final`, and calls with a single receiver method are devirtualized. We also remove uncallable methods and assign non-conflicting slots to interface methods using a graph-coloring algorithm. The results of some class casts and `instanceof` operations can also be determined statically using these results.

Our analysis keeps separate the set of *mentioned* and *instantiated* classes. Although the program can contain type-checks on and method-invocations of abstract, interface, or otherwise uninstantiated classes, every object in the heap must belong to one of the instantiated class types. The size of the set of instantiated classes is quite small for a typical Java program, and over half of the benchmarks in SPECjvm98 have less than 256 instantiated class types.[4] We use this information to replace the class pointer in the object header, which identifies the type of the object, with a one-byte *index* into a small lookup table. The `jess`, `javac`, and `jack` benchmarks require more than one byte of index, but a two byte index amply suffices in these three cases.

3.3.2 Bitwidth Analysis

We use a flow-sensitive interprocedural combined value-propagation and bitwidth analysis to find constant values, unread and constant fields, and to reduce field sizes where possible. Since almost all types in Java are signed (with the exception of the 16-bit `char`), we must be able to describe bitwidths of both negative and positive numbers, which we do by splitting the set of values into negative, zero, and positive parts, and describing the bitwidth of each individually.

We abstract non-singleton sets of integer values into a tuple $\langle m, p\rangle$ where $m \geq 1 + \lfloor \log_2 N \rfloor$ for all negative N in the set, and $p \geq 1 + \lfloor \log_2 N \rfloor$ for positive N. We use $m = p = 0$ to represent the constant zero. Some combination rules for arithmetic operations are shown in Figure 3-5. The rules for simple arithmetic operators should

[4]Note that *all* have more than 256 *total* class types.

$$
\begin{array}{rcl}
-\langle m, p\rangle & = & \langle p, m\rangle \\
\langle m_l, p_l\rangle + \langle m_r, p_r\rangle & = & \langle 1+\max(m_l, m_r), 1+\max(p_l, p_r)\rangle \\
\langle m_l, p_l\rangle \times \langle m_r, p_r\rangle & = & \left\langle \begin{array}{c} \max(m_l+p_r, p_l+m_r), \\ \max(m_l+m_r, p_l+p_r) \end{array} \right\rangle \\
\langle 0, p_l\rangle \wedge \langle 0, p_r\rangle & = & \langle 0, \min(p_l, p_r)\rangle \\
\langle m_l, p_l\rangle \wedge \langle m_r, p_r\rangle & = & \langle \max(m_l, m_r), \max(p_l, p_r)\rangle
\end{array}
$$

Figure 3-5: Some combination rules for bitwidth analysis of arithmetic and bitwise-logical operators. Note that the penultimate entry is a special-case rule that only applies if the neither of the arguments can be negative.

be self-evident upon examination (adding two N bit integers yields at most an $N+1$-bit integer, for example) although care must be taken to ensure that combinations of negative and positive integers are handled correctly. Our implementation contains additional rules giving it greater precision for common special cases, such as multiplication by a one-bit quantity, division by a constant, and (as the figure shows) bitwise operations on positive numbers.

Treatment of Fields

Dataflow on this bitwidth lattice is performed on the entire Java program interprocedurally. The analysis is *field-based* [119]: for each field f in class X, the analysis uses the abstract analysis value $X.f$ to represent all of the values in the f field of instances of X. The analysis therefore models an assignment to f in any instance of X as an assignment to the corresponding analysis value $X.f$.[5] The result of the analysis is a bitwidth specification for each variable and field in the program. We also identify constant variables and fields; we replace reads of constant fields with their constant value and eliminate the field. Fields for which no reads are found (even if writes are present) are also eliminated.[6]

Other Details

Our analysis handles method calls by merging the lattice values of the method parameters at the call site with the formal parameters of the method. Similarly, the return value of the method is propagated back to all call-sites. Our compiler's intermediate representation handles thrown exceptions by treating the method return value as a tuple, and the call site as a conditional branch. The "normal return value" is assigned and the first branch taken on a normal method return, and the "exceptional return

[5]An obvious extension is to use pointer analysis to discriminate between fields allocated at different program points.

[6]Note that checks which may throw exceptions on reads and writes are preserved.

Benchmark	**total fields**	**unread**	**constant**	**% alloc'ed space saved**
compress	298	75	31	2.5%
jess	485	91	43	9.9%
raytrace	341	75	30	0.0%
db	286	75	35	0.0%
javac	531	85	34	0.6%
mpegaudio	286	75	35	1.4%
mtrt	341	75	30	0.0%
jack	378	77	31	10.2%

Table 3.1: Number of unused and constant fields in SPEC benchmarks, and the savings realized (in % of total dynamic allocated bytes) by removing them.

value" is assigned and the second branch taken when an exception is thrown from the method.

Our implementation of this analysis is actually context-sensitive, with a user-defined context length. All results presented here were obtained with the context set to zero; we saw no clear benefit from 1- or 2-deep calling contexts, and the increase in analysis time was considerable.

Space does not permit us to describe the remaining details of the full analysis, including the extension of the value lattice to handle the full range of Java types, the class hierarchy, `null` and `String` constants, and fixed-length arrays. We refer the interested reader to [18] for an exhaustive description of the intraprocedural analysis.

In Table 3.1 we show the number of unread and constant fields found by this analysis in our benchmark set. Table 3.2 shows the space reductions due to bitwidth analysis and field reduction using our byte packing strategy.

3.3.3 Definite Initialization Analysis

Java field semantics dictate that uninitialized fields must have the value zero (or `null`, for pointer fields). It may seem, then, that the starting lattice value for every integer field should be $\mathbf{0}$. This starting value, however, prevents us from finding nonzero field constants in the program: a simple initialization statement like `x=5` will assign `x` the value $\mathbf{0} \sqcap \mathbf{5}$, which is not equal to $\mathbf{5}$![7]

We perform a *definite initialization* analysis to remedy this problem and restore precision to our analysis. For example, with only constructor $\mathtt{A}_1$ in the following code, field `f` will get the lattice value $\mathbf{5}$:

```
public class A {
    int f;
    A₁(...)  { f = 5; }
    A₂(...)  { /* no assignment to f */ }
}
```

[7]On the SCC lattice of [201], $\mathbf{0} \sqcap \mathbf{5} = \top$ (but see footnote 8).

	static field bits		% alloc'ed
Benchmark	**before**	**after**	**space saved**
compress	7591	5430	3.0%
jess	13349	10634	30.1%
raytrace	7467	5296	0.9%
db	6777	4983	0.3%
javac	11560	8161	5.4%
mpegaudio	6777	4983	1.5%
mtrt	7467	5296	0.9%
jack	8356	6037	17.2%

Table 3.2: Number of field bits in SPEC benchmarks statically removed due to bitwidth analysis, and the dynamic savings (in % of total allocated bytes) of field bitwidth reduction using byte packing.

Without constructor $\texttt{A}_2$ in the class, we say that field `f` is *definitely initialized* because every constructor of `A` assigns a value to `f` before returning or calling an unsafe method. Adding constructor $\texttt{A}_2$ allows the default **0** value of `f` to be seen; `f` is then no longer definitely initialized.

We actually allow the constructor great flexibility with regard to definite initialization; it is free to call any method which does not read `A.f` before finally executing a definite initializer. We construct a mapping from methods to all fields which they may read, in a flow-insensitive manner, and compute a transitive closure of this map over the call graph to determine a "safe set" of methods which the constructor may call before a definite initialization of `f`. As long as control flow may not pass to a method not in the safe set before `f` is written, then `f` is definitely initialized.

When performing bitwidth analysis, definitely-initialized fields are allowed to start at $\perp$ in the dataflow lattice.[8] All other fields must start at value **0**, which will make it impossible for the field to represent a nonzero constant value. The results of the definite initialization analysis are also used when profiling mostly-constant fields, as described in the next section.

3.3.4 Profiling Mostly-Constant Fields

To inform the static specialization and field externalization transformations, we instrument a profiling build of the code to determine which fields are *mostly-constant*. Our implementation builds one binary per examined constant, that is, one binary to look for "mostly-zero" fields, a separate binary to look for fields which are usually "one", a third binary to look for fields commonly "two", and so forth. We built eleven binaries for each benchmark, looking for field default values in the interval $[-5, 5]$. For pointer fields, we only look for `null` as a default value. It should be stressed that our use of multiple separate binaries was solely for ease of implementation, and is not

[8]We use $\perp$ for "nothing known" and $\top$ for "under-constrained"; another segment of the compiler community commonly reverses these definitions.

Benchmark	**Field**	**always-zero bytes**		**field bytes dyn. alloc'd**	**zero %**	**benchmark total dyn. alloc'n**
compress	Hashtable$Entry.next	3,552	/	7,148	49.7%	105MB
	String.offset	3,180	/	3,500	90.9%	
jess	jess.Token.negcnt	7,573,616	/	7,573,616	100.0%	252MB
	jess.Value.floatval	5,688,080	/	10,170,640	55.9%	
raytrace	Point.z	4,101,328	/	17,464,188	23.5%	126MB
	Point.x	3,291,076	/	17,464,188	18.8%	
db	String.offset	508,204	/	508,524	99.9%	73MB
	Vector.capacityIncrement	62,548	/	62,548	100.0%	
javac	String.offset	3,735,388	/	3,847,816	97.1%	161MB
	Statement.labels	578,608	/	578,688	100.0%	
mpegaudio	Hashtable$Entry.next	3,616	/	7,336	49.3%	666kB
	String.offset	2,352	/	2,672	88.0%	
jack	String.offset	7,442,956	/	7,443,276	100.0%	178MB
	Hashtable$Enumerator.type	5,288,364	/	5,288,364	100.0%	

Table 3.3: Representative "mostly-zero" fields found in SPEC benchmarks.

an inherent limitation of the technique.

Our instrumentation pass starts by adding a counter per class to record the number of times each exact class type is instantiated. We also add per-field counters which are incremented the first time a non-N value is stored into a certain field.[9] By comparing the number of times the class (thus field) is instantiated and the number of times the field is set to a non-N value, we can determine the amount of memory recoverable by applying a "mostly-N" transformation to the field, whether static specialization or field externalization. We use this potential savings to guide our selection of fields for static specialization, using the field and default value which the profile indicates will yield the largest gain. If static specialization isn't an option, the proportion of non-N fields helps indicate whether externalization is likely to result in a net savings; see Section 3.4.2 for further discussion.

There is one last detail to attend to: when looking for nonzero N values, the default zero value of uninitialized fields becomes a problem. For these cases, we use the definite-initialization analysis described in the previous section to increment the "non-N" counter on any path where the field in question is not definitely initialized.

Table 3.3 presents some representative "mostly-zero" fields which our profiling technique identifies in the SPEC benchmarks.

3.3.5 Finding Subclass-Final Fields

Our static specialization transformation can only be applied to what we call *subclass-final* fields. Subclass-finality is a less strict but similar constraint to Java's `final` modifier. We do a single-pass analysis to determine subclass-finality, using the results from the bitwidth analysis to improve our precision.[10]

A *subclass-final* field `f` of a class `A` can be written to from any method of a *subclass* of `A`, as well as in any constructor of `A`. In each write, the receiver's type must be a subtype of `A`, except inside `A`'s constructors, where the receiver may also be the method's `this` parameter. Other writes are disallowed. Unlike fields marked

[9]Note that implementing this counter requires storing an additional bit per field during profiling to record whether a non-N value has been seen previously.

[10]By using analysis rather than relying on programmer specification, the author need not restrict *all* users of their code in order to obtain maximum efficiency for *some* constrained uses of it.

with Java's `final` modifier, multiple writes to `f` are permitted, as long as each write satisfies the above constraints.

Subclass-finality matches the requirements of the static specialization transformation. Since we always insert a "big" version of the class between the specialized class and its children, subclasses can write to the field present in objects of the "big" type without restriction. We need only restrict writes which occur in the class proper.

Our analysis constructs the set of subclass-final fields by finding its dual, the set of *non*-subclass-final fields. We scan every method and collect all fields with illegal writes; all fields found are added to the set of non-subclass-final fields.

3.3.6 Constructor Classification

The final requirement to enable static specialization is to identify constructors which always initialize certain fields in a given way. In particular, we wish to find constructors which always give fields statically–known-constant values, as well as constructors which initialize fields with simple functions of their input parameters. The first case enables us to unconditionally replace an instantiated class with a smaller split version; the second case allows us to wrap the constructor in an appropriate conditional to enable the creation of the small version when dynamically possible.

This analysis builds upon our previous results. In a single pass over the constructor, we merge the values written to a selected subclass-final field, treating `Param`N as an abstract value for the Nth constructor parameter. We treat any call to a `this()` constructor as if it were inlined. By the properties of subclass-final fields, we know that all writes to the field are to the `this` object and that there are no bad writes to the field outside of the constructor. If the merged value at the end of the pass is a `Param` value or a constant equal to the desired "default" value of the selected field, then we can statically specialize on the field for calls to this particular constructor. Further, we rule out specialization on any otherwise-suitable fields for which there is not at least one callable constructor amenable to static specialization.

3.4 Implementation Issues

In this section we will talk briefly about some of the practical issues arising in an implementation of our space-saving techniques.

3.4.1 Byte Packing

A typical Java implementation may waste large amounts of space by aligning fields for the most efficient memory access. Fields are often aligned to their widths (a 4-byte field will be placed at an address which is an even multiple of 4, for example), and the object as a whole is often placed on a double-word boundary. Our implementation places object fields at the nearest byte boundary, although the information provided by our bitwidth analysis is sufficient to *bit*-pack the fields in the object when space is truly at a premium. Preliminary investigation indicated that the amount of additional space gained by bit-packing is typically only a few percent, because there aren't enough sub-byte fields to fill the space "wasted" by byte alignment.[11]

Some architectures penalize unaligned accesses to fields. It is worthwhile to *attempt* to align fields to their preferred alignment while not allowing this alignment to cause the object size to grow. Further, there are often *forced* alignment constraints on (for example) pointers. Our Java runtime uses a conservative garbage collector; its efficiency decreases markedly if pointers are not word-aligned.[12]

Our "byte-packing" heuristic achieves tight packing of fields while respecting forced alignments. Packing proceeds recursively through superclasses, and returns a list of free-space intervals available between the fields of the superclass. The algorithm first places all *forced-alignment* fields in the class, from largest to smallest. The aim is for the alignment-induced spaces left by the large fields to be fillable by the following smaller fields.

When there are no more forced-alignment fields, we attempt to allocate fields on their "preferred" alignment boundaries, largest first. At this stage fields are not allowed to introduce an alignment gap at the end of the object. If their preferred alignment does not allow them to be placed flush against the last field of the object, they are skipped.

Finally, when there are no more fields satisfying preferred-alignments, we allocate the *smallest* available field at the lowest possible byte boundary. The aim is that the small fields will fill space and nudge the end of the object out so that a larger field may be allocated on its preferred alignment. After each field is placed, we begin again by attempting to place fields on preferred boundaries.

We have observed that this heuristic strategy works well in practice, and the penalties for occasionally placing an unaligned non-pointer field were not seen to have a material adverse effect on performance (see Section 3.5.3).

3.4.2 External Hashtable Implementation

The implementation of the hashtable used for field and hash/lock externalization can dramatically affect the space savings possible with these transformations. The overhead of dynamically-allocated buckets and the required *next* pointers makes separate chaining impractical as a hashtable implementation technique. Open-addressing implementations are preferable: in addition to the stored data, all that is necessary is a key value and the empty space required to limit the load factor. A load factor of two-thirds and one-word keys and values yield an average space consumption of three words per field. This implementation breaks even when the mostly-zero fields identified are zero over 66% of the time. This break-even point is compared to the profiling data to allow our field externalization transformation to intelligently choose

[11]Note also that "bit-packing" may lead to the loss of atomicity on concurrent writes to adjacent fields packed within a byte, typically the processor's smallest atomic write size. An escape analysis would be sufficient to ensure that fields accessed from differing threads are not packed within the same atomic unit.

[12]This pointer alignment restriction means that objects have to be word-aligned as well.

targeted fields.

Key-size reduction is an important component of the implementation: a naïve approach would combine a one-word reference to the virtual-container object and a one-word field identifier for a two-word key. The large key will shift the break-even point up so that only fields which are 82% zero will profit. Instead, we can offset the object reference (up to the limit of its size) by small integers to discriminate the externalized fields of the object, yielding a single-word key.

Our implementation puts a weak reference to the object in the hashtable, enabling the garbage collector to remove unneeded entries.

3.4.3 Class Loading and Reflection

We conducted this research using the MIT FLEX compiler infrastructure,[13] which is a whole-program static compiler. Although the analyses as described reflect this compilation model, it would be straightforward to use *extant analysis* [184] to apply transformations to only the closed-world portions of a program which used dynamic class loading. The space allocated to the class index could be updated during garbage collection as new classes are discovered. Concurrent profiling could actually expose more opportunities for space compression in a JIT environment. Finally, our various transformations need not be exposed to the program if the reflection implementation is carefully written.

3.5 Experimental Results

We have implemented all of the analyses and transformations described in this paper in FLEX. We measure the effectiveness of our optimizations by using FLEX to analyze the SPECjvm98 benchmarks and apply our transformations, then measuring the resulting space savings and performance. All benchmarks were run with the full input size on a dual-processor 900 MHz Pentium III running Debian Linux.

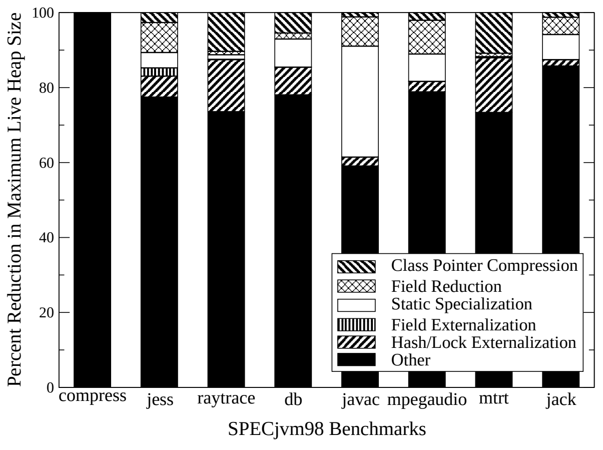
3.5.1 Memory Savings

To evaluate the effectiveness of our technique at reducing the amount of memory required to execute the program, we first ran an instrumented version of each application with no space optimizations. We used this instrumented version to compute the maximum amount of live data on the heap at any point during the execution. We then ran an instrumented version of our program after each stage of optimization. These versions enabled us to calculate the amount by which each technique reduced the size of the live heap data.[14]

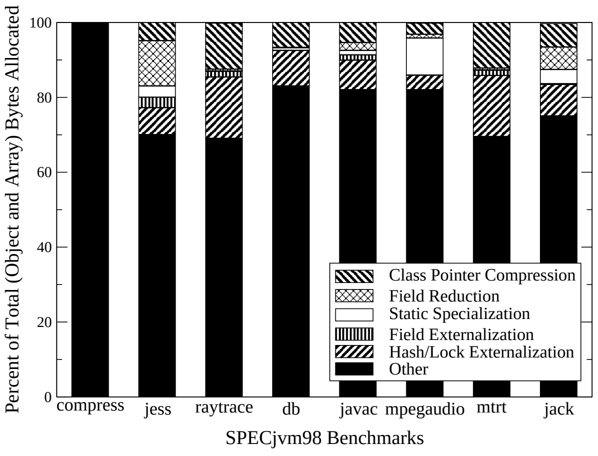
Figure 3-6a presents the total space savings. This figure contains a bar for each application, with the bar broken down into categories that indicate the percentage

[13]Available from `http://flexc.lcs.mit.edu/`.

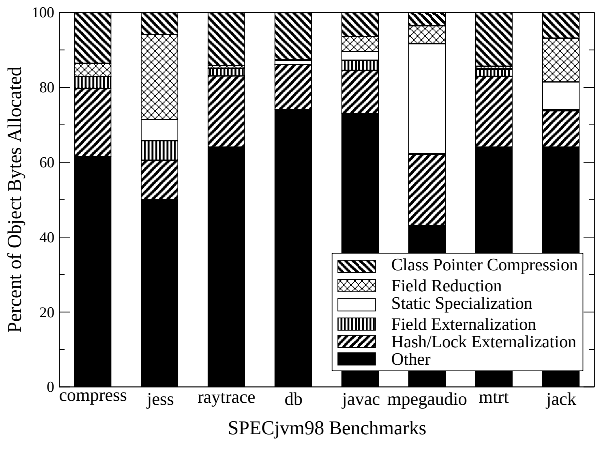
[14]The instrumented versions collect all non-live data before each allocation, so that our computed maximum heap sizes are accurate.



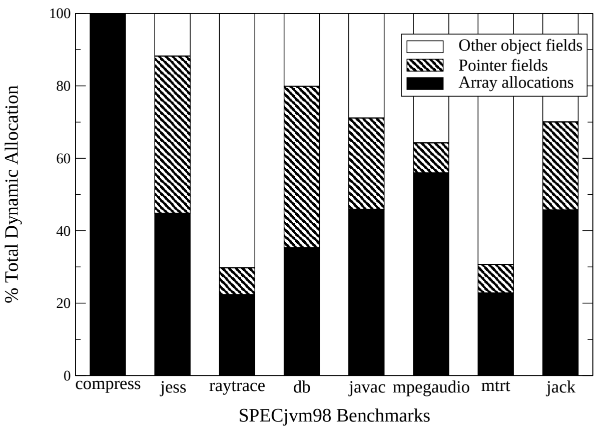
(a) Reduction in the maximum live heap achieved with our transformations.



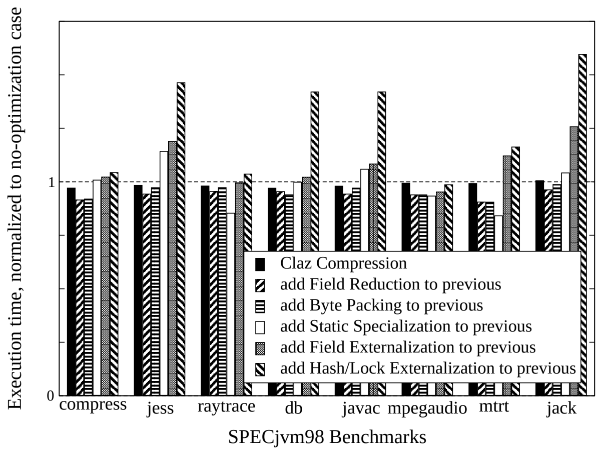
(b) Cumulative reduction in dynamic allocation achieved with our transformations.



(c) Reduction in non-array dynamic allocation achieved with our transformations.



(d) Pre-transformation allocation breakdown between arrays and objects, with allocations attributable to fields of pointer type split out.



(e) Runtime performance of space optimizations.

Figure 3-6: Experimental results of space optimization transformations.

of live data from the original unoptimized execution that we were able to eliminate with each optimization. The black section of each bar indicates the amount of live heap data remaining after all optimizations. We obtain as much as 40% reduction in live data on the `javac` benchmark, with almost all of this reduction coming from our bitwidth-driven field reductions and static specialization. In fact we obtain more than 15% reduction on all of the "object-oriented" benchmarks. The `compress` benchmark allocates a small number of very large arrays, limiting the optimization opportunities discoverable by our analysis. Likewise, the `raytrace` and `mtrt` benchmarks make heavy use of floating-point numbers, limiting the applicability of our integer bitwidth analysis. However, these raytracing benchmarks allocate a large number of small arrays to represent vectors and matrices, and so our header optimizations still allow us to reduce the maximum live data size by over 20%.

We also used an instrumented executable to determine the total amount of memory allocated during the entire execution of the program, in both the optimized and unoptimized versions. Reducing this total allocation decreases the load on the garbage collector. Figure 3-6b presents the space savings according to this metric. Comparison to the previous figure reveals that long-lived objects provide proportionally more opportunities for optimization.

3.5.2 Objects Versus Arrays

The majority of our optimizations are designed to optimize object fields rather than arrays. For context, we present numbers that characterize the reductions in total allocation for objects only, rather than for both objects and arrays. Figure 3-6c presents space savings numbers for objects alone, omitting any storage required for arrays. Figure 3-6d explains the difference by showing how the total program allocation for each benchmark is broken down into array and object allocations. The reason for our poor performance on `compress` is now obvious—a few large uncompressible integer arrays account for over 99% of the total space allocated.

3.5.3 Execution Times

We next evaluate the execution time impact of applying our space optimizations. Figure 3-6e presents the normalized execution times of each benchmark after the application of our sequence of optimizations. These numbers show that the first several optimizations (class pointer compression, field reduction, and byte packing) typically reduce the execution times, while the remainder (static specialization, field externalization, and hash/lock externalization) generate modest increases in the execution times. The speedup is due to reduced GC times, despite the indirection and misalignment costs. Static specialization's virtualization of fields is responsible for its slowdown; it is likely that an optimized speculatively-inlined implementation of the field accessors which it adds to the program would improve its performance. Field externalization (including hash/lock externalization) causes the expected penalty for hashtable lookup; note that synchronization elimination would greatly reduce the cost of hash/lock externalization in the four cases where the overhead is unreasonable.

3.6 Related Work

Many researchers have focused on the problem of reducing the amount of header space required to represent Java locks [25, 156, 2]. The vast majority of programs do not use the lock associated with every object in its full generality, so it is possible to develop improved algorithms optimized for the common case. The idea is to represent the lock with the minimum amount of state (typically a bit) required to support the common usage pattern of an acquire followed by a release, and to back off to a more elaborate scheme only when the thread exhibits a more complex pattern such as nested locking. The primary focus has been on improving performance rather than on reducing space; however, many of the algorithms also eliminate the need to store the complicated locking objects required to support the most general lock usage pattern possible in a Java program. These techniques typically reduce the lock space overhead to 24 header bits [25]; Bacon et al. in [24] show speed improvements from header-size reduction, in agreement with the results presented here.

Research on escape analysis and related analyses can enable the compiler to find objects whose locks are never acquired [12, 37, 202, 58, 172, 191]. This information can enable the compiler to remove the space reserved for synchronization support in these objects. Our hash/lock removal algorithm uses a totally dynamic approach based on our field externalization mechanism.

Several researchers have used bitwidth analysis to reduce the size of the generated circuits for compilers that generate hardware implementations of programs written in C or similar programming languages [15, 18, 174, 186, 50].

Dieckmann and Hölzle have performed an in-depth analysis of the memory allocation behavior of Java programs [81]. Although space is not their primary focus, their study does quantify the space overhead associated with the use of a two-word header and of 8-byte alignment. In general, our measurements of the memory system behavior of Java programs broadly agree with their measurements.

Sweeney and Tip [192] did a study of dead members of C++ programs, which is similar to the unread field elimination done by our bitwidth analysis. However, they fail to identify constant members, as our analysis algorithm can. Further, our results show that unread and constant field elimination is very dependent on the coding style of a particular application. The collection of techniques we have presented here gives much more consistent savings over a wide range of benchmarks.

Aggarwal and Randall [5] described an array bounds check removal method using *related fields*. This work attempted to discover fields, such as `Vector.size`, which are guaranteed to be less than or equal to the length of some array, for example, the backing array stored in `Vector.data`. Tests against the related field could then provide information about bounds checks on accesses to the array. This technique could be used to infer additional bitwidth information on related fields from our analysis.

Marinov and O'Callahan have presented Object Equality Profiling [151], a technique which identifies when several instances of an object may be safely merged to a single representative instance. The merging which is suggested is an orthogonal memory-saving measure which could be used in addition to the ones described here.

Zhang and Gupta describe a runtime technique that recognizes two special cases when an integer or a pointer field in a designated C data structure may be compressed [210]. For all but two of their benchmarks, their heap savings (on these benchmarks, an average of 27%) are entirely due to a pointer compression techique which is orthogonal to the transformations described in this paper. The techniques could be combined for greater savings.

3.7 Conclusions

We have presented a set of techniques for reducing the memory consumption of object-oriented programs. Our techniques include program analyses to detect unused, constant, or overly-wide fields, and transformations to eliminate fields with common default values or usage patterns. These techniques apply equally well to both user-defined fields and fields implicit in the runtime's object header, and can reduce the maximum heap required for a program by as much as 40%. Our experimental results from our fully-implemented system validate the opportunity for space savings on typical object oriented programs.

Chapter 4

Pointer and Escape Analysis for Multithreaded Programs

4.1 Introduction

Multithreading is a key structuring technique for modern software. Programmers use multiple threads of control for many reasons: to build responsive servers that communicate with multiple parallel clients [157], to exploit the parallelism in shared-memory multiprocessors [55], to produce sophisticated user interfaces [163], and to enable a variety of other program structuring approaches [118].

Research in program analysis has traditionally focused on sequential programs [154]; extensions for multithreaded programs have usually assumed a block structured, parbegin/parend form of multithreading in which a parent thread starts several parallel threads, then immediately blocks waiting for them to finish [135, 173]. But the standard form of multithreading supported by languages such as Java and threads packages such as POSIX threads is unstructured — child threads execute independently of their parent threads. The software structuring techniques described above are designed to work with this form of multithreading, as are many recommended design patterns [142]. But because the lifetimes of child threads potentially exceed the lifetime of their starting procedure, unstructured multithreading significantly complicates the interprocedural analysis of multithreaded programs.

4.1.1 Analysis Algorithm

This chapter presents a new combined pointer and escape analysis for multithreaded programs, including programs with unstructured forms of multithreading. The algorithm is based on a new abstraction, *parallel interaction graphs*, which maintain precise points-to, escape, and action ordering information for objects accessed by multiple threads. Unlike previous escape analysis abstractions, parallel interaction graphs enable the algorithm to analyze the interactions between parallel threads. The analysis can therefore capture objects that are accessed by multiple threads but do not escape a given multithreaded computation. It can also fully characterize the points-to relationships for objects accessed by multiple parallel threads.

Because parallel interaction graphs characterize all of the potential interactions of the analyzed method or thread with its callers and other parallel threads, the resulting analysis is compositional at both the method and thread levels — it analyzes each method or thread once to produce a single general analysis result that can be specialized for use in any context.[1] Finally, the combination of points-to and escape information in the same abstraction enables the algorithm to analyze only part of the program, with the analysis result becoming more precise as more of the program is analyzed.

4.1.2 Application to Region-Based Allocation

We have implemented our analysis in the MIT Flex compiler for Java. The information that it produces has many potential applications in compiler optimizations, software engineering, and as a foundation for further program analysis. This chapter presents our experience using the analysis to optimize and check safety conditions for programs that use region-based allocation constructs instead of relying on garbage collection. Region-based allocation allows the program to run (a potentially multithreaded) computation in the context of a specific allocation region. All objects created by the computation are allocated in the region and deallocated when the computation finishes. To avoid dangling references, the implementation must ensure that the objects in the region do not outlive the associated computation. One standard way to achieve this goal is to dynamically check that the program never attempts to create a reference from one object to another object allocated in a region with a shorter lifetime [38]. If the program does attempt to create such a reference, the implementation refuses to create the reference and throws an exception. Unfortunately, this approach imposes dynamic checking overhead and introduces a new failure mode for programs that use region-based allocation.

We have used our analysis to statically verify that our multithreaded benchmark programs use region-based allocation correctly. It therefore provides a safety guarantee to the programmer and enables the compiler to eliminate the dynamic region reference checks. We also found that intrathread analysis alone is not powerful enough — the algorithm must analyze the interactions between parallel threads to verify the correct use of region-based allocation.

We also used our analysis for the more traditional purpose of synchronization elimination. While our algorithm is quite effective at enabling this optimization, for our multithreaded benchmarks, the interthread analysis provides little additional benefit over the standard intrathread analysis.

4.1.3 Contributions

This chapter makes the following contributions:

[1]Recursive methods or recursively generated threads may require an iterative algorithm that may analyze methods or threads in the same strongly connected component multiple times to reach a fixed point.

- **Abstraction:** It presents a new abstraction, parallel interaction graphs, for the combined pointer and escape analysis of programs with unstructured multithreading.
- **Analysis:** It presents a new algorithm for analyzing multithreaded programs. The algorithm is compositional and analyzes interactions between parallel threads.
- **Region-Based Allocation:** It presents our experience using the analysis to statically verify that programs correctly use region-based allocation constructs. The benefits include providing a safety guarantee for the program and eliminating the overhead of dynamic region reference checks.

The remainder of the chapter is structured as follows. Section 4.2 presents an example that illustrates how the algorithm works. Section 4.3 presents the abstractions that the analysis uses, while Section 4.4 presents the analysis algorithm and Section 4.5 discusses the analysis uses. We discuss experimental results in Section 4.6, related work in Section 4.7, and conclude in Section 4.8.

4.2 Example

We next present a simple example that illustrates how the analysis works.

4.2.1 Structure of the Parallel Computation

Figure 4-1 presents a multithreaded Java program that computes the Fibonacci number of its input. The `Task` class implements a parallel divide and conquer algorithm for this computation. Each `Task` stores an `Integer` object in its `source` field as input and produces a new `Integer` object in its `target` field as output.[2]

This program illustrates several common patterns for multithreaded programs. First, it uses threads to implement parallel computations. Second, when a thread starts its execution, it points to objects that hold the input data for its computation. Finally, when the computation finishes, it writes references to its result objects into its thread object for the parent computation to read.

4.2.2 Regions and Memory Management

As the computation runs, it continually allocates new `Task` objects for the parallel subcomputations and new `Integer` objects to hold their inputs and outputs. The lifetimes of these objects are contained in the lifetime of the Fibonacci computation, and die when this computation finishes. A standard memory management system would not exploit this property. The `Task` and `Integer` objects would be allocated out

[2]This program uses the standard Java thread creation mechanism. The statement `t1.start()` creates a new parallel thread of control. This new thread of control then invokes the `run` method of the `Task` class on the `t1` object. This `start/run` linkage is the standard way to execute new threads in Java.

```
class main {
  public static void main(String args[]) {
    int i = Integer.parseInt(args[0]);
    Fib f = new Fib(i);
    Region r = new Region();
    r.enter(f);
  }
}
class Fib implements Runnable {
  int source;

  Fib(int i) { source = i; }

  public void run() {
    Task t = new Task(new Integer(source));
    t.start();
    try {
      t.join();
    } catch (Exception e) { System.out.println(e); }
    System.out.println(t.target.toString());
  }
}
class Task extends Thread {
  public Integer source;
  public Integer target;

  Task(Integer s) { source = s; }

  public void run() {
    int v = source.intValue();
    if (v <= 1) {
      target = source;
    } else {
      Task t1 = new Task(new Integer(v-1));
      Task t2 = new Task(new Integer(v-2));
      t1.start();
      t2.start();
      try {
        t1.join();
        t2.join();
      } catch (Exception e) { System.out.println(e); }
      int x = t1.target.intValue();
      int y = t2.target.intValue();
      target = new Integer(x + y);
    }
  }
}
```

Figure 4-1: Multithreaded Fibonacci Example

of the garbage-collected heap, increasing the memory consumption rate, the garbage collection frequency, and therefore the garbage collection overhead.

Region-based allocation provides an attractive alternative. Instead of allocating all objects out of a single garbage-collected heap, region-based approaches allow the program to create multiple memory regions, then allocate each object in a specific region. When the program no longer needs any of the objects in the region, it deallocates all of the objects in that region without garbage collection.

Researchers have proposed many different region-based allocation systems. Our example (and our implemented system) uses the approach standardized in the Real-Time Java specification [38]. Before the `main` program invokes the Fibonacci computation, it creates a new memory region `r`. The statement `r.enter(f)` executes the `run` method of the `f` object (and all of the methods or threads that it executes) in the context of the new region `r`. When one of the threads in this computation creates a new object, the object is allocated in the region `r`. When the entire multithreaded computation terminates, all of the objects in the region are deallocated without garbage collection. The `Task` and `Integer` objects are therefore managed independently of the garbage collected heap and do not increase the garbage collection frequency or overhead. Region-based allocation is an attractive alternative to garbage collection because it exploits the correspondence between the lifetimes of objects and the lifetimes of computations to deliver a more efficient memory management mechanism.

4.2.3 Regions and Dangling Reference Checks

One potential problem with region-based allocation is the possibility of dangling references. If an object whose lifetime exceeds the region's lifetime refers to an object allocated inside the region, any use of the reference after the region is deallocated will access potentially recycled garbage, violating the memory safety of the program. The Real-Time Java specification eliminates this possibility as follows. It allows the computation to create a hierarchy of nested regions and ensures that no parent region is deallocated before one of its child regions. Each region is associated with a (potentially multithreaded) computation; the objects in the region are deallocated when its computation terminates and the objects in all of its child regions have been deallocated. The implementation dynamically checks all assignments to object fields to ensure that the program never attempts to create a reference that goes down the hierarchy from an object in an ancestor region to an object in a child region. If the program does attempt to create such a reference, the check fails. The implementation prevents the assignment from taking place and throws an exception.

While these checks ensure the memory safety of the execution, they impose additional execution time overhead and introduce a new failure mode for the software. Our goal is to analyze the program and statically verify that the checks never fail. Such an analysis would enable the compiler to eliminate all of the dynamic region checks. It would also provide the programmer with a guarantee that the program would never throw an exception because a check failed.

4.2.4 Analysis in the Example

We use a generalized escape analysis to determine whether any object allocated in a given region escapes the computation associated with the region. If none of the objects escape, the program will never attempt to create a dangling reference and the compiler can eliminate all of the checks. The algorithm first performs an intrathread, interprocedural analysis to derive a parallel interaction graph at the end of each method. Figures 4-2 and 4-3 present the analysis results for the `run` methods in the `Fib` and `Task` classes, respectively.

Points-to Graphs

The first component of the parallel interaction graph is the points-to graph. The nodes in this graph represent objects; the edges represent references between objects. There are two kinds of edges: *inside* edges, which represent references created within the analyzed part of the program (for Figure 4-2, the sequential computation of the `Fib.run` method), and *outside* edges, which represent references read from objects potentially accessed outside the analyzed part of the program. In our figures, solid lines denote inside edges and dashed lines denote outside edges.

There are also several kinds of nodes. *Inside* nodes represent objects created within the analyzed part of the program. There is one inside node for each object creation site in the program; that node represents all objects created at that site. *Parameter* nodes represent objects passed as parameters to the currently analyzed method; *load* nodes represent objects accessed by reading a reference in an object potentially accessed outside the analyzed part of the program. Together, the parameter and load nodes make up the set of *outside* nodes. In our figures, solid circles denote inside nodes and dashed circles denote outside nodes.

In Figure 4-2, nodes `1` and `4` are outside nodes. Node `1` represents the `this` parameter of the method, while node `4` represents the object whose reference is loaded by the expression `t.target` at line 2 of the example at the end of the `Fib.run` method. Nodes `2` and `3` are inside nodes, and denote the `Task` and `Integer` objects created in the statement `Task t = new Task(new Integer(source))` at line `1` of the example.

Started Thread Information

The parallel interaction graph contains information about which threads were started by the analyzed part of the program. In Figure 4-2, node `2` represents the started `Task` thread that implements the entire Fibonacci computation. In Figure 4-3, nodes `8` and `11` represent the two threads that implement the parallel subtasks in the computation. The interthread analysis uses the started thread information when it computes the interactions between the current thread and threads that execute in parallel with the current thread.

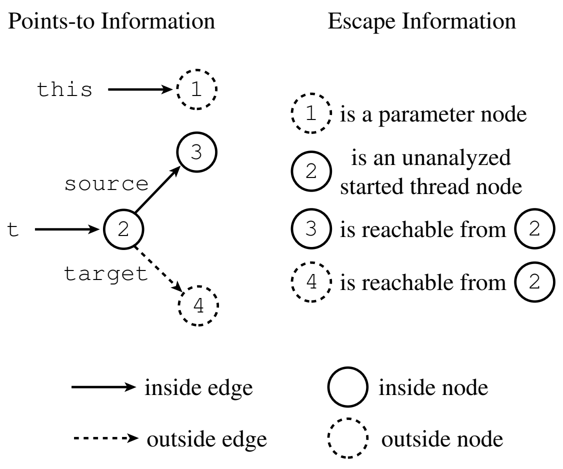


Figure 4-2: Analysis Result for `Fib.run`

Escape Information

The parallel interaction graph contains information about how objects escape the analyzed part of the program to be accessed by the unanalyzed part. A node escapes if it is a parameter node or represents an unanalyzed thread started within the analyzed part of the program. It also escapes if it is reachable from an escaped node. In Figure 4-2, node `1` escapes because it is passed as a parameter, while nodes `3` and `4` escape because they are reachable from the unanalyzed thread node `2`.

4.2.5 Interthread Analysis

Previously proposed escape analyses treat threads very conservatively — if an object is reachable from a thread object, the analyses assume that it has permanently escaped [35, 37, 58, 202]. Our algorithm, however, analyzes the interactions between threads to recapture objects accessed by multiple threads. The foundation of the interthread analysis is the construction of two mappings μ_1 and μ_2 between the nodes of the parallel interaction graphs of the parent and child threads. Each outside node is mapped to another node if the two nodes represent the same object during the analysis. The mappings are used to combine the parallel interaction graph from the child thread into the parallel interaction graph from the parent thread. The result is a new parallel interaction graph that summarizes the parallel execution of the two threads.

Figure 4-4 presents the mappings from the interthread analysis of `Fib.run` and the `Task.run` method for the thread that `Fib.run` starts. The algorithm computes these mappings as follows:

- **Initialization:** Inside the `Fib.run` method, node `2` represents the started `Task` thread. Inside the `Task.run` method, node `5` represents the same started thread. The algorithm therefore initializes μ_2 to map node `5` to node `2`.

- **Matching `target` edges:** The analysis of the `Task.run` method creates inside

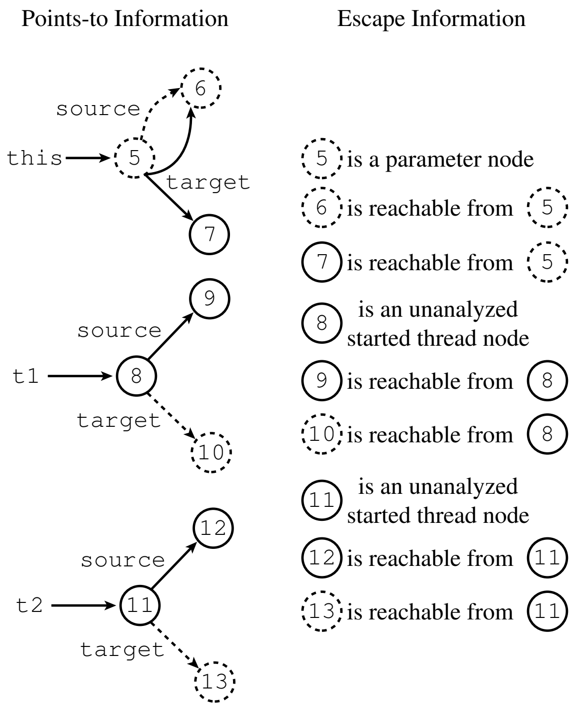


Figure 4-3: Analysis Result for `Task.run`

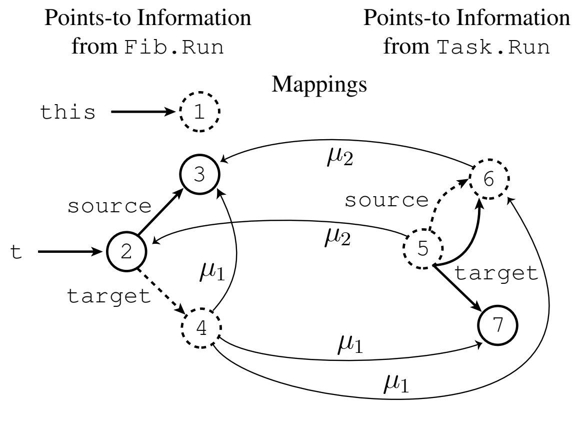

Points-to Information from Fib.Run
Points-to Information from Task.Run
Mappings
this
t
source
target
μ_1
μ_2

Figure 4-4: Mappings for Interthread Analysis of `Fib.run` and `Task.run`

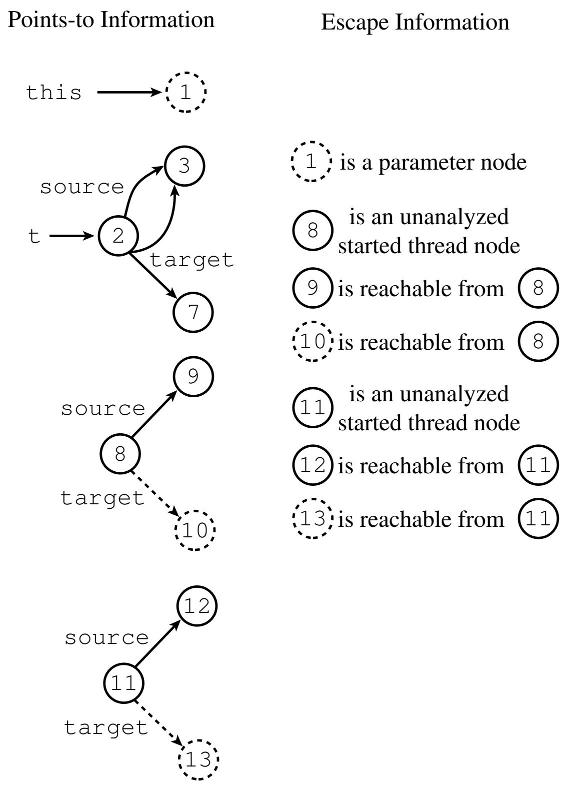


Figure 4-5: Analysis Result After First Interthread Analysis

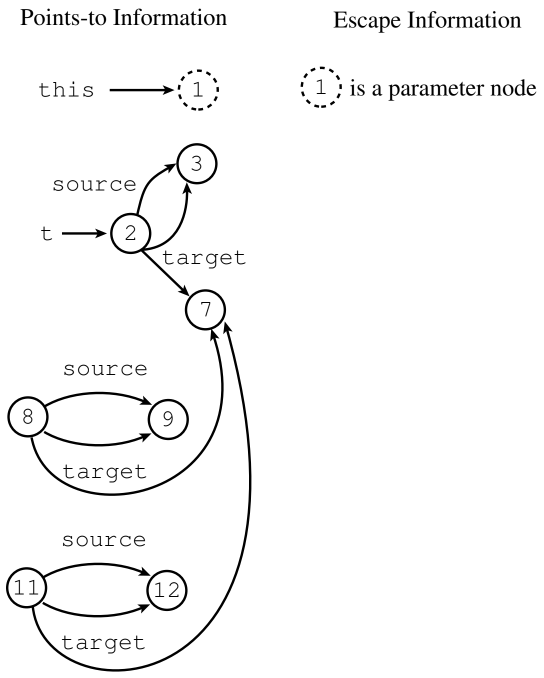


Figure 4-6: Final Analysis Result for `Fib.run`

edges from node 5 to nodes 6 and 7. These edges have the label `target`, and represent references between the corresponding `Task` and `Integer` objects during the execution of the `Task.run` method.

The `Fib.run` method reads these references to obtain the result of the `Task.run` method. The outside edge from node 2 to node 4 represents these references during the analysis of the `Fib.run` method. The analysis therefore matches the outside edge from the `Fib.run` method (from node 2 to node 4) against the inside edges from the `Task.run` method to compute that node 4 represents the same objects as nodes 6 and 7. The result is that μ_1 maps node 4 to nodes 6 and 7.

- **Matching `source` edges:** The analysis of the `Fib.run` method creates an inside edge from node 2 to node 3. This edge has the label `source`, and represents a reference between the corresponding `Task` and `Integer` objects during the execution of the `Fib.run` method.

  The `Task.run` method reads this reference to obtain its input. The outside edge from node 5 to node 6 represents this reference during the analysis of the `Task.run` method. The interthread analysis therefore matches the outside edge from the `Task.run` method (from node 5 to node 6) against the inside edge from the `Fib.run` method (from node 2 to node 3) to compute that node 6 represents the same objects as node 3. The result is that μ_2 maps node 6 to node 3.

- **Transitive Mapping:** Because μ_1 maps node 4 to node 6 and μ_2 maps node 6 to node 3, the analysis computes that node 4 represents the same object as node 3. The result is that μ_1 maps node 4 to node 3.

Note that the matching process models interactions in which one thread reads references created by the other thread. Because the threads execute in parallel, the matching is symmetric.

The analysis uses μ_1 and μ_2 to combine the two parallel interaction graphs and obtain a new graph that represents the combined effect of the two threads. Figure 4-5 presents this graph, which the analysis computes as follows:

- **Edge Projections:** The analysis projects the edges through the mappings to augment nodes from one parallel interaction graph with edges from the other graph. In our example, the analysis projects the inside edge from node 5 to node 6 through μ_2 to generate new inside edges from node 2 to nodes 3 and 7. It also generates other edges involving outside nodes, but removes these edges during the simplification step.

- **Graph Combination:** The analysis combines the two graphs, omitting the outside node that represents the `this` parameter of the started thread (node 5 in our example).

- **Simplification:** The analysis removes all outside edges from captured nodes, all outside nodes that are not reachable from a parameter node or unanalyzed started thread node, and all inside nodes that are not reachable from a live variable, parameter node, or unanalyzed started thread node.

In our example, the analysis recaptures the (now analyzed) thread node `2`. Nodes `3` and `7` are also captured *even though they are reachable from a thread node.* The analysis removes nodes `4` and `6` in the new graph because they are not reachable from a parameter node or unanalyzed thread node. Note that because the interactions with the thread nodes `8` and `11` have not yet been analyzed, those nodes and all nodes reachable from them escape.

Because our example program uses recursively generated parallelism, the analysis must perform a fixed point computation during the interthread analysis. Figure 4-6 presents the final parallel interaction graph from the end of the `Fib.run` method, which is the result of this fixed point analysis. The analysis has recaptured all of the inside nodes, including the task nodes. Because none of the objects represented by these nodes escapes the computation of the `Fib.run` method, its execution in a new region will not violate the region referencing constraints.

4.3 Analysis Abstraction

We next formally present the abstraction (parallel interaction graphs) that the analysis uses. In addition to the points-to and escape information discussed in Section 4.2, parallel interaction graphs can also represent ordering information between actions (such as synchronization actions) from parent and child threads. This ordering information enables the analysis to determine when thread start events temporally separate actions of parent and child threads. This information may, for example, enable the analysis to determine that a parent thread performs all of its synchronizations on a given object before a child thread starts its execution and synchronizes on the object. To simplify the presentation, we assume that the program does not use static class variables, all the methods are analyzable and none of the methods returns a result. Our implemented analysis correctly handles all of these aspects [189].

4.3.1 Object Representation

The analysis represents the objects that the program manipulates using a set $n \in N$ of nodes, which is the disjoint union of the set N_I of inside nodes and the set N_O of outside nodes. The set of thread nodes $N_T \subseteq N_I$ represents thread objects. The set of outside nodes is the disjoint union of the set N_L of load nodes and the set N_P of parameter nodes. There is also a set $\mathtt{f} \in \mathtt{F}$ of fields in objects, a set $\mathtt{v} \in \mathtt{V}$ of local and parameter variables, and a set $\mathtt{l} \in \mathtt{L} \subseteq \mathtt{V}$ of local variables.

4.3.2 Points-To Escape Graphs

A points-to escape graph is a triple $\langle O, I, e \rangle$, where

- $O \subseteq N \times \mathtt{F} \times N_L$ is a set of outside edges. We use the notation $O(n_1, \mathtt{f}) = \{n_2 | \langle n_1, \mathtt{f}, n_2 \rangle \in O\}$.

- $I \subseteq (N \times \mathtt{F} \times N) \cup (\mathtt{V} \times N)$ is a set of inside edges. We use the notation $I(\mathtt{v}) = \{n | \langle \mathtt{v}, n \rangle \in I\}$, $I(n_1, \mathtt{f}) = \{n_2 | \langle n_1, \mathtt{f}, n_2 \rangle \in I\}$.
- $e : N \to \mathcal{P}(N)$ is an escape function that records the escape information for each node.[3] A node escapes if it is reachable from a parameter node or from a node that represents an unanalyzed parallel thread.

The escape function must satisfy the invariant that if n_1 points to n_2, then n_2 escapes in at least all of the ways that n_1 escapes. When the analysis adds an edge to the points-to escape graph, it updates the escape function so that it satisfies this invariant. We define the concepts of escaped and captured nodes as follows:

- escaped($\langle O, I, e \rangle, n$) if $e(n) \neq \emptyset$
- captured($\langle O, I, e \rangle, n$) if $e(n) = \emptyset$

4.3.3 Parallel Interaction Graphs

A parallel interaction graph is a tuple $\langle \langle O, I, e \rangle, \tau, \alpha, \pi \rangle$:

- The thread set $\tau \subseteq N$ represents the set of unanalyzed thread objects started by the analyzed computation.
- The action set α records the set of actions executed by the analyzed computation. Each synchronization action $\langle \mathtt{sync}, n_1, n_2 \rangle \in \alpha$ has a node n_1 that represents the object on which the action was performed and a node n_2 that represents the thread that performed the action. If the action was performed by the current thread, n_2 is the dummy current thread node $n_{\mathrm{CT}} \in N_T$. Our implementation can also record actions such as reading an object, writing an object, or invoking a given method on an object. It is straightforward to generalize the concept of actions to include actions performed on multiple objects.
- The action order π records ordering information between the actions of the current thread and threads that execute in parallel with the current thread.
  - $\langle \langle \mathtt{sync}, n_1, n_2 \rangle, n \rangle \in \pi$ if the synchronization action $\langle \mathtt{sync}, n_1, n_2 \rangle$ may have happened after one of the threads represented by n started executing. In this case, the actions of a thread represented by n may conflict with the action.
  - $\langle \langle n_1, \mathtt{f}, n_2 \rangle, n \rangle \in \pi$ if a reference represented by the outside edge $\langle n_1, \mathtt{f}, n_2 \rangle$ may have been read after one of the threads represented by n started executing. In this case, the outside edge may represent a reference written by a thread represented by n.

We use the notation $\pi @ n = \{a | \langle a, n \rangle \in \pi\}$ to denote the set of actions and outside edges in π that may occur in parallel with a thread represented by n.

[3]Here $\mathcal{P}(N)$ is the set of all subsets of N, so that $e(n)$ is the set of nodes through which n escapes.

4.4 Analysis Algorithm

For each program point, the algorithm computes a parallel interaction graph for the current analysis scope at that point. For the intraprocedural analysis, the analysis scope is the currently analyzed method up to that point. The interprocedural analysis extends the scope to include the (transitively) called methods; the interthread analysis further extends the scope to include the started threads.

We next present the analysis, identifying the program representation, the different phases, and the key algorithms in the interprocedural and interthread phases.

4.4.1 Program Representation

The algorithm represents the computation of each method using a control flow graph. We assume the program has been preprocessed so that all statements relevant to the analysis are either a copy statement $\mathtt{l} = \mathtt{v}$, a load statement $\mathtt{l}_1 = \mathtt{l}_2.\mathtt{f}$, a store statement $\mathtt{l}_1.\mathtt{f} = \mathtt{l}_2$, a synchronization statement $\mathtt{l}.\mathtt{acquire}()$ or $\mathtt{l}.\mathtt{release}()$, an object creation statement $\mathtt{l} = \mathtt{new\ cl}$, a method invocation statement $\mathtt{l}_0.\mathtt{op}(\mathtt{l}_1, \ldots, \mathtt{l}_k)$, or a thread start statement $\mathtt{l}.\mathtt{start}()$.

The control flow graph for each method `op` starts with an enter statement $\mathtt{enter}_{\mathtt{op}}$ and ends with an exit statement $\mathtt{exit}_{\mathtt{op}}$.

4.4.2 Intraprocedural Analysis

The intraprocedural analysis is a forward dataflow analysis that propagates parallel interaction graphs through the statements of the method's control flow graph. Each method is analyzed under the assumption that the parameters are *maximally unaliased*, i.e., point to different objects. For a method with formal parameters $\mathtt{v}_0, \ldots, \mathtt{v}_n$, the initial parallel interaction graph at the entry point of the method is $\langle\langle\emptyset, \{\langle \mathtt{v}_i, n_{\mathtt{v}_i}\rangle\}, \lambda n.\text{if } n = n_{\mathtt{v}_i} \text{ then } \{n\} \text{ else } \emptyset\rangle, \emptyset, \emptyset, \emptyset\rangle$, where $n_{\mathtt{v}_i}$ is the parameter node for parameter $\mathtt{v}_i$. If the method is invoked in a context where some of the parameters may point to the same object, the interprocedural analysis described below in Section 4.4.4 merges parameter nodes to conservatively model the effect of the aliasing.

The transfer function $\langle G', \tau', \alpha', \pi'\rangle = [\![\mathtt{st}]\!](\langle G, \tau, \alpha, \pi\rangle)$ models the effect of each statement `st` on the current parallel interaction graph. Figure 4-7 graphically presents the rules that determine the new points-to graphs for the different basic statements. Each row in this figure contains four items: a statement, a graphical representation of existing edges, a graphical representation of the existing edges plus the new edges that the statement generates, and a set of side conditions. The interpretation of each row is that whenever the points-to escape graph contains the existing edges and the side conditions are satisfied, the transfer function for the statement generates the new edges. Assignments to a variable kill existing edges from that variable; assignments to fields of objects leave existing edges in place.

In addition to updating the outside and inside edge sets, the transfer function also updates the the escape function e to ensure that if n_1 points to n_2, then n_2 escapes

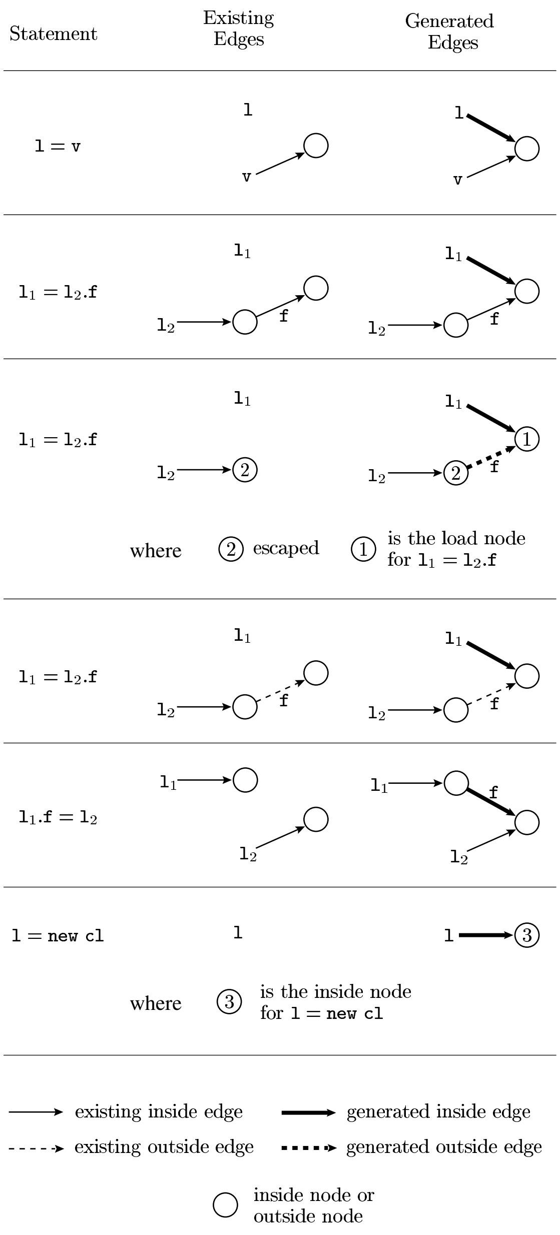


Figure 4-7: Generated Edges for Basic Statements

$$
\begin{aligned}
\tau' &= \tau \cup I(\mathtt{l}) \\
e'(n) &= \begin{cases} e(n) \cup \{n'\} & \text{if } n' \in I(\mathtt{l}) \text{ and} \\ & \quad n \text{ is reachable in } O \cup I \text{ from } n' \\ e(n) & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 4-8: Transfer Function for `l.start()`

$$\begin{aligned}\alpha' &= \alpha \cup \{\texttt{sync}\} \times I(\texttt{l}) \times \{n_{\mathrm{CT}}\} \\ \pi' &= \pi \cup (\{\texttt{sync}\} \times I(\texttt{l}) \times \{n_{\mathrm{CT}}\}) \times \tau\end{aligned}$$

Figure 4-9: Transfer Function for `l.acquire()` and `l.release()`

in at least all of the ways that n_1 escapes. Except for load statements, the transfer functions leave τ, α, and π unchanged. For a load statement $\texttt{l}_1 = \texttt{l}_2.\texttt{f}$ the transfer function updates the action order π to record that any new outside edges may be created in parallel with the threads modeled by the nodes in τ (here n_L is the load node for $\texttt{l}_1 = \texttt{l}_2.\texttt{f}$):

$$\pi' = \pi \cup \{\langle n_1, \texttt{f}, n_L \rangle | n_1 \in I(\texttt{l}_2) \wedge \text{escaped}(\langle O, I, e\rangle, n_1)\} \times \tau$$

Figure 4-8 presents the transfer function for an `l.start()` statement, which adds the started thread nodes to τ and updates the escape function. Figure 4-9 presents the transfer function for synchronization statements, which add the corresponding synchronization actions into α and record the actions as executing in parallel with all of the nodes in τ. At control-flow merges, the confluence operation takes the union of the inside and outside edges, thread sets, actions, and action orders.

4.4.3 Mappings

Mappings $\mu : N \to \mathcal{P}(\mathcal{N})$ implement the substitutions that take place when combining parallel interaction graphs. During the interprocedural analysis, for example, a parameter node from a callee is mapped to all of the nodes at the call site that may represent the corresponding actual parameter. Given an analysis component ξ, $\xi[\mu]$ denotes the component after replacing each node n in ξ with $\mu(n)$:[4]

$$\begin{aligned}\tau[\mu] &= \textstyle\bigcup_{n\in\tau} \mu(n) \\ O[\mu] &= \textstyle\bigcup_{\langle n,\texttt{f},n_L\rangle\in O} \mu(n) \times \{\texttt{f}\} \times \{n_L\} \\ I[\mu] &= \textstyle\bigcup_{\langle n_1,\texttt{f},n_2\rangle\in I} \mu(n_1) \times \{\texttt{f}\} \times \mu(n_2) \cup \bigcup_{\langle \texttt{v},n\rangle\in I}\{\texttt{v}\} \times \mu(n) \\ \alpha[\mu] &= \textstyle\bigcup_{\langle \texttt{sync},n_1,n_2\rangle\in\alpha}\{\texttt{sync}\} \times \mu(n_1) \times \mu(n_2) \\ \pi[\mu] &= \textstyle\bigcup_{\langle\langle \texttt{sync},n_1,n_2\rangle,n\rangle\in\pi}(\{\texttt{sync}\} \times \mu(n_1) \times \mu(n_2)) \times \mu(n) \cup \\ &\quad \textstyle\bigcup_{\langle\langle n_1,\texttt{f},n_2\rangle,n\rangle\in\pi}(\mu(n_1) \times \{\texttt{f}\} \times \mu(n_2)) \times \mu(n)\end{aligned}$$

4.4.4 Interprocedural Analysis

The interprocedural analysis computes a transfer function for each method invocation statement. We assume a method invocation site of the form $\texttt{l}_0.\texttt{op}(\texttt{l}_1, \ldots, \texttt{l}_k)$, a potentially invoked method `op` with formal parameters $\texttt{v}_0, \ldots, \texttt{v}_k$ with corresponding pa-

[4]The only exception is in the definition of $O[\mu]$ where we do not substitute the load node n_L that constitutes the end point of an outside edge $\langle n, \texttt{f}, n_L\rangle$.

rameter nodes $n_{\mathtt{v}_0}, n_{\mathtt{v}_1}, \ldots, n_{\mathtt{v}_k}$, a parallel interaction graph $\langle\langle O_1, I_1, e_1\rangle, \tau_1, \alpha_1, \pi_1\rangle$ at the program point before the method invocation site, and a graph $\langle\langle O_2, I_2, e_2\rangle, \tau_2, \alpha_2, \pi_2\rangle$ from the `exit` statement of `op`. The interprocedural analysis has two steps. It first computes a mapping μ for the outside nodes from the callee. It then uses μ to combine the two parallel interaction graphs to obtain the parallel interaction graph at the program point immediately after the method invocation. The analysis computes μ as the least fixed point of the following constraints:

$$I_1(l_i) \subseteq \mu(n_{\mathtt{v}_i}), \forall i \in \{0, 1, \ldots k\} \tag{4.1}$$

$$\frac{\langle n_1, \mathtt{f}, n_2\rangle \in O_2, \langle n_3, \mathtt{f}, n_4\rangle \in I_1, n_3 \in \mu(n_1)}{n_4 \in \mu(n_2)} \tag{4.2}$$

$$\frac{\begin{array}{c}\langle n_1, \mathtt{f}, n_2\rangle \in O_2, \langle n_3, \mathtt{f}, n_4\rangle \in I_2,\\ \mu(n_1) \cap \mu(n_3) \neq \emptyset, n_1 \neq n_3\end{array}}{\mu(n_4) \cup \{n_4\} \subseteq \mu(n_2)} \tag{4.3}$$

The first constraint initializes μ; the next two constraints extend μ. Constraint 4.1 maps each parameter node from the callee to the nodes from the caller that represent the actual parameters at the call site. Constraint 4.2 matches outside edges read by the callee against corresponding inside edges from the caller. Constraint 4.3 matches outside edges from the callee against inside edges from the callee to model aliasing between callee nodes.

The algorithm next extends μ to μ' to ensure that all nodes from the callee (except the parameter nodes) appear in the new parallel interaction graph:

$$\mu'(n) = \begin{cases} \mu(n) & \text{if } n \in N_P \\ \mu(n) \cup \{n\} & \text{otherwise} \end{cases}$$

The algorithm computes the new parallel interaction graph $\langle\langle O', I', e'\rangle, \tau', \alpha', \pi'\rangle$ at the program point after the method invocation as follows:

$$\begin{array}{ll} O' = O_1 \cup O_2[\mu'] & I' = I_1 \cup (I_2 - V \times N)[\mu'] \\ \tau' = \tau_1 \cup \tau_2[\mu'] & \alpha' = \alpha_1 \cup \alpha_2[\mu'] \\ \multicolumn{2}{c}{\pi' = \pi_1 \cup \pi_2[\mu'] \cup (O_2[\mu'] \cup \alpha_2[\mu']) \times \tau_1} \end{array}$$

It computes the new escape function e' as the union of the escape function e_1 before the method invocation and the expansion of the escape function e_2 from the callee through μ'. More formally, the following constraints define the new escape function e' as

$$e_1(n) \subseteq e'(n) \qquad \frac{n_2 \in \mu'(n_1)}{(e_2(n_1) - N_P)[\mu'] \subseteq e'(n_2)}$$

propagated over the edges from $O' \cup I'$. After the interprocedural analysis, reachability from the parameter nodes of the callee is no longer relevant for the escape function, hence the set difference in the second initialization constraint. We have a proof that this interprocedural analysis produces to a parallel interaction graph that is at least as conservative as the one that would be obtained by inlining the callee and performing the intraprocedural analysis as in section 4.4.2 [189].

Finally, we simplify the resulting parallel interaction graph by removing superfluous nodes and edges. We remove all load nodes n_L such that $e'(n_L) = \emptyset$ from the graph; such load nodes do not represent any concrete object. We also remove all all outside edges $\langle n_1, \mathtt{f}, n_2 \rangle$ that start from a captured node n_1 (where $e'(n_1) = \emptyset$); such outside edges do not represent any concrete reference. Finally, we remove all nodes that are not reachable from a live variable, parameter node, or unanalyzed started thread node from τ'.

Because of dynamic dispatch, a single method invocation site may invoke several different methods. The transfer function therefore merges the parallel interaction graphs from all potentially invoked methods to derive the parallel interaction graph at the point after the method invocation site. The current implementation obtains this call graph information using a variant of a cartesian product type analysis [4], but it can use any conservative approximation to the dynamic call graph.

The analysis uses a worklist algorithm to solve the combined intraprocedural and interprocedural dataflow equations. A bottom-up analysis of the program yields the full result with one analysis per strongly connected component of the call graph. Within strongly connected components, the algorithm iterates to a fixed point.

4.4.5 Thread Interaction

Interactions between threads take place between a starter thread (a thread that starts a parallel thread) and a startee thread (the thread that is started). The interaction algorithm is given the parallel interaction graph $\langle\langle O, I, e\rangle, \tau, \alpha, \pi\rangle$ from a program point in the starter thread, a node n_T that represents the startee thread, and a `run` method that runs when the thread object represented by n_T starts. The parallel interaction graph associated with the `exit` statement of the `run` method is $\langle\langle O_2, I_2, e_2\rangle, \tau_2, \alpha_2, \pi_2\rangle$. The result of the thread interaction algorithm is a parallel interaction graph $\langle\langle O', I', e'\rangle, \tau', \alpha', \pi'\rangle$ that models all the interactions between the execution of the starter thread (up to its corresponding program point) and the entire startee thread. This result conservatively models all possible interleavings of the two threads.

The algorithm has two steps. It first computes two mappings μ_1, μ_2, where μ_1 maps outside nodes from the starter and μ_2 maps outside nodes from the startee. It then uses μ_1 and μ_2 to combine the two parallel interaction into a single parallel interaction graph that reflects the interactions between the two threads. The algorithm computes μ_1 and μ_2 as the least fixed point of the following constraints:

$$n_T \in \mu_2(n_{\mathtt{v}_0}), n_T \in \mu_2(n_{\mathrm{CT}}) \tag{4.4}$$

$$\frac{\langle n_1, \mathtt{f}, n_2\rangle \in O_i, \langle n_3, \mathtt{f}, n_4\rangle \in I_j, n_3 \in \mu_i(n_1)}{n_4 \in \mu_i(n_2)} \tag{4.5}$$

$$\frac{\begin{array}{c}\langle n_1, \mathtt{f}, n_2\rangle \in O_i, \langle n_3, \mathtt{f}, n_4\rangle \in I_i,\\ \mu_i(n_1) \cap \mu_i(n_3) \neq \emptyset, n_1 \neq n_3\end{array}}{\mu_i(n_4) \cup \{n_4\} \subseteq \mu_i(n_2)} \tag{4.6}$$

$$\frac{\langle n_1, \mathtt{f}, n_2\rangle \in I_i, \langle n_3, \mathtt{f}, n_4\rangle \in O_j, n_3 \in \mu_i(n_1)}{n_2 \in \mu_j(n_4)} \tag{4.7}$$

$$\frac{n_2 \in \mu_i(n_1), n_3 \in \mu_j(n_2)}{n_3 \in \mu_i(n_1)} \tag{4.8}$$

Here $n_{\mathtt{v}_0}$ is the parameter node associated with the single parameter of the `run` method – the `this` pointer – and n_{CT} is the dummy current thread node. Also, $I_1 = I$ and $O_1 = O \cap (\pi @ n_T)$. Note that the algorithm computes interactions only for outside edges from the starter thread that represent references read after the startee thread starts.

Unlike the caller/callee interaction, where the execution of the caller is suspended during the execution of the callee, in the starter/startee interaction, both threads execute in parallel, producing a more complicated set of statement interleavings. The interthread analysis must therefore model a richer set of potential interactions in which each thread can read edges created by the other thread. The interthread analysis therefore uses two mappings (one for each thread) instead of just one mapping. It also augments the constraints to reflect the potential interactions.

In the same style as in the interprocedural analysis, the algorithm first initializes the mappings μ'_1, μ'_2 to extend μ_1 and μ_2, respectively. Each node from the two initial parallel interaction graphs (except $n_{\mathtt{v}_0}$) will appear in the new parallel interaction graph:

$$\mu'_1(n) = \mu_1(n) \cup \{n\}$$
$$\mu'_2(n) = \begin{cases} \mu_2(n) & \text{if } n = n_{\mathtt{v}_0} \\ \mu_2(n) \cup \{n\} & \text{otherwise} \end{cases}$$

The algorithm uses μ'_1 and μ'_2 to compute the resulting parallel interaction graph as follows:

$$\begin{array}{ll} O' = O[\mu'_1] \cup O_2[\mu'_2] & I' = I[\mu'_1] \cup (I_2 - V \times N)[\mu'_2] \\ \tau' = \tau[\mu'_1] \cup \tau_2[\mu'_2] & \alpha' = \alpha[\mu'_1] \cup \alpha_2[\mu'_2] \end{array}$$
$$\begin{array}{rcl} \pi' & = & \pi[\mu'_1] \cup \pi_2[\mu'_2] \cup \\ & & (O_2[\mu'_2] \cup \alpha_2[\mu'_2]) \times \tau[\mu'_1] \cup \pi @ n_T[\mu'_1] \times \tau_2[\mu'_2] \end{array}$$

In addition to combining the action orderings from the starter and startee, the algorithm also updates the new action order π' to reflect the following ordering relationships:

- All actions and outside edges from the startee occur in parallel with all of the starter's threads, and

- All actions and outside edges from the starter thread that occur in parallel with the startee thread also occur in parallel with all of the threads that the startee starts.

The new escape function e' is the union of the escape function e from the starter and the escape function e_2 from the startee, expanded through μ_1 and μ_2, respectively.

More formally, the escape function e' is initialized by the following two constraints

$$\frac{n_2 \in \mu_1(n_1)}{e(n_1)[\mu_1] \subseteq e'(n_2)} \qquad \frac{n_2 \in \mu_2(n_1)}{(e_2(n_1) - N_P)[\mu_2] \subseteq e'(n_2)}$$

and propagated over the edges from $O' \cup I'$.

4.4.6 Interthread Analysis

The interthread analysis uses a fixed-point algorithm to obtain a single parallel interaction graph that reflects the interactions between all of the parallel threads. The algorithm repeatedly chooses a node $n_T \in \tau$, retrieves the analysis result from the `exit` node of the corresponding `run` method,[5] then uses the thread interaction algorithm presented above in Section 4.4.5 to compute the interactions between the analyzed threads and the thread represented by n_T and combine the two parallel interaction graphs into a new graph. Once the algorithm reaches a fixed point, it removes all nodes in N_T from the escape function — the final graph already models all of the possible interactions that may affect nodes that escape only via unanalyzed thread nodes. The analysis may therefore recapture thread nodes that escaped before the interthread analysis. For example, if a thread node does not escape via a parameter node, it is captured after the interthread analysis. Finally the algorithm enhances the efficiency and precision of the analysis by removing superfluous nodes and edges using the same simplification method as in the interprocedural analysis.

As presented, the algorithm assumes that each node $n \in \tau$ represents multiple instances of the corresponding thread. Our implementation improves the precision of the analysis by tracking whether each node represents a single thread or multiple threads. For nodes that represent a single thread, the algorithm computes the interactions just once, adjusting the new action order π' to record that the outside edges and actions from the startee thread do not occur in parallel with the node n that represents the startee thread. For nodes that represent multiple threads, the algorithm repeatedly computes the interactions until it reaches a fixed point.

4.4.7 Resolving Outside Nodes

It is possible to augment the algorithm so that it records, for each outside node, all of the inside nodes that it represents during the analysis of the entire program. This information allows the algorithm to go back to the analysis results generated at the various program points and resolve each outside node to the set of inside nodes that it represents during the analysis. In the absence of nodes that escape via unanalyzed

[5]The algorithm uses the type information to determine which class contains this `run` method. For inside nodes, this approach is exact. For outside nodes, the algorithm uses class hierarchy analysis to find a set of classes that may contain the `run` method. The algorithm computes the interactions with each of the possible `run` methods, then merges the results. In practice, τ almost always contains inside nodes only — the common coding practice is to create and start threads in the same method.

threads or methods, this enables the algorithm to obtain complete, precise points-to information even for analysis results that contain outside nodes.

4.5 Analysis Uses

We next discuss how we use the analysis results to perform two optimizations: region reference check elimination and synchronization elimination.

4.5.1 Region Reference Check Elimination

The analysis eliminates region reference checks by verifying that no object allocated in a given region escapes the computation that executes in the context of that region. In our system, all such computations are invoked via the execution of a statement of the form `r.enter(t)`. This statement causes the the `run` method of the thread `t` to execute in the context of the memory region `r`. The analysis first locates all of these `run` methods. It then analyzes each `run` method, performing both the intrathread and interthread analysis, and checks that none of the inside nodes in the analysis result escape. If none of these inside nodes escape, all of the objects allocated inside the region are inaccessible when the computation terminates. All of the region reference checks will therefore succeed and can be removed.

4.5.2 Synchronization Elimination

The synchronization elimination algorithm uses the results of the interthread analysis to find captured objects whose synchronization operations can be removed. Like previous synchronization elimination algorithms, our algorithm uses the intrathread analysis results to remove synchronizations on objects that do not escape the thread that created them. Unlike previous synchronization elimination algorithms, our algorithm also analyzes the interactions between parallel threads. It then uses the action set α and the action ordering relation π to eliminate synchronizations on objects with synchronizations from multiple threads.

The analysis proceeds as follows. For each node n that is captured after the interthread analysis, it examines π to find all threads t that execute in parallel with a synchronization on n. It then examines the action set α to determine if t also synchronizes on n. If none of the parallel threads t synchronize on n, the compiler can remove all synchronizations on the objects that n represents. Even if multiple threads synchronize on these objects, the analysis has determined that the synchronizations are temporally separated by thread start events and therefore redundant.

4.6 Experimental Results

We have implemented our combined pointer and escape analysis algorithm in the MIT Flex compiler system, a static compiler for Java. We used the analysis information for synchronization elimination and elimination of dynamic region reference checks.

We present experimental results for a set of multithreaded benchmark programs. In general, these programs fall into two categories: web servers and scientific computations. The web servers include Http, an http server, and Quote, a stock quote server. Both of these applications were written by others and posted on the Internet. Our scientific programs include Barnes and Water, two complete scientific applications that have appeared in other benchmark sets, including the SPLASH-2 parallel computing benchmark set [205]. We also present results for two synthetic benchmarks, Tree and Array, that use object field assignment heavily. These benchmarks are designed to obtain the maximum possible benefit from region reference check elimination.

4.6.1 Methodology

We first modified the benchmark programs to use region-based allocation. The web servers create a new thread to service each new connection. The modified versions use a separate region for each connection. The scientific programs execute a sequence of interleaved serial and parallel phases. The modified versions use a separate region for each parallel phase. The result is that all of the modified benchmarks allocate long-lived shared objects in the garbage-collected heap and short-lived objects in regions. The modifications were relatively straightforward to perform, but it was difficult to evaluate the correctness of the modifications without the static analysis. The web servers were particularly problematic since they heavily use the Java libraries. Without the static analysis it was not clear to us that the libraries would work correctly with region-based allocation. For Http, Quote, Tree, and Array, the interprocedural analysis alone was able to verify the correct use of region-based allocation and enable the elimination of all dynamic region checks. Barnes and Water required the interthread analysis to eliminate the checks — interprocedural analysis alone was unable to verify the correct use of region-based allocation.

We used the MIT Flex compiler to generate a C implementation of each benchmark, then used gcc to compile the program to an x86 executable. We ran the Http and Quote servers on a 400 MHz Pentium II running Linux, with the clients running on an 866 MHz Pentium III running Linux. The two machines were connected with their own private 100 Mbit/sec Ethernet. We ran Water, Barnes, Tree, and Array on an 866 MHz Pentium III running Linux.

4.6.2 Results

Figure 4-10 presents the program sizes and analysis times. The synchronization elimination algorithm analyzes the entire program, while the region check algorithm analyzes only the `run` methods and the methods that they (transitively) invoke. The synchronization elimination analysis therefore takes significantly more time than the region analysis. The backend time is the time required to produce an executable once the analysis has finished. All times are in seconds. Figure 4-11 presents the number of synchronizations for the Original version with no analysis, the Interprocedural version with interprocedural analysis only, and the Interthread version with both interprocedural and interthread analysis. For this optimization, the interthread

Program	Bytecode instructions	Analysis time [s] for removing checks	Analysis time [s] for removing syncs	Backend time [s]
Tree	10,970	0.5	15.9	41.1
Array	10,896	0.6	16.9	42.2
Water	17,675	11.3	56.1	66.0
Barnes	15,945	6.9	94.2	54.8
Http	14,313	17.1	38.3	73.8
Quote	14,039	16.9	41.4	61.4

Figure 4-10: Program Sizes and Analysis Times

Program	Original version	Optimized version: Interprocedural	Optimized version: Interthread
Tree	59	43	43
Array	59	43	43
Water	2,367,193	919,575	919,575
Barnes	2,838,720	678,355	678,355
Http	67,268	8,460	7,406
Quote	268,913	200,650	198,610

Figure 4-11: Number of Synchronization Operations

Program	Standard	Checks	No Checks
Tree	6.5	16.8	7.0
Array	8.2	43.4	8.3
Water	9.6	9.7	8.1
Barnes	8.4	7.6	6.7
Http	4.5	5.3	5.2
Quote	11.7	11.3	11.3

Figure 4-12: Execution Times for Benchmarks

Program	Number of Objects in Heap	Number of Objects in Regions
Tree	184	65,534
Array	183	8
Water	20,755	3,110,675
Barnes	17,622	2,121,167
Http	12,228	62,062
Quote	21,785	121,350

Figure 4-13: Allocation Statistics for Benchmarks

analysis produces almost no additional benefit over the interprocedural analysis. Figure 4-12 presents the execution times of the benchmarks. The Standard version allocates all objects in the garbage-collected heap and does not use region-based allocation. The Checks version uses region-based allocation with all of the dynamic checks. The No Checks version uses region-based allocation with the analysis eliminating all dynamic checks. None of the versions uses the synchronization elimination optimization. Check elimination produces substantial performance improvements for Tree and Array and modest performance improvements for Water and Barnes. The running times of Http and Quote are dominated by thread creation and operating system overheads, so check elimination provides basically no performance increase. Figure 4-13 presents the number of objects allocated in the garbage-collected heap and the number allocated in regions. The vast majority of the objects are allocated in regions.

4.6.3 Discussion

Our applications use regions in one of two ways. The servers allocate a new region for each connection. The region holds the new objects required to service the connection. Examples of such objects include `String` objects that hold responses sent to clients and iterator objects used to find requested data. The scientific programs use regions for auxiliary objects that structure the parallel computation. These objects include the `Thread` objects required to generate the parallel computation and objects that hold values produced by intermediate calculations.

In general, eliminating region checks provides modest performance improvements. We therefore view the primary value of the analysis in this context as helping the programmer to use regions correctly. We expect the analysis to be especially useful in situations (such as our web servers) when the programmer may not have complete confidence in his or her detailed knowledge of the program's object usage patterns.

4.7 Related Work

We discuss several areas of related work: analysis of multithreaded programs, escape analysis for multithreaded programs, and region-based allocation.

4.7.1 Analysis of Multithreaded Programs

The analysis of multithreaded programs is a relatively unexplored field [167]. There is an awareness that multithreading significantly complicates program analysis but a full range of standard techniques have yet to emerge. Grunwald and Srinivasan present a dataflow analysis framework for reaching definitions for explicitly parallel programs [112], and Knoop, Steffen and Vollmer present an efficient dataflow analysis framework for bit-vector problems such as liveness, reachability and available expressions [135]. Both frameworks are designed for programs with structured, parbegin/parend concurrency and are intraprocedural. We view the main contributions of the reserach presented in this chapter as largely orthogonal to this previous research. In particular, our main contribution center on abstractions and algorithms for the interprocedural and compositional analysis of programs with unstructured multithreading. We also focus on problems, pointer and escape analysis, that do not fit within either framework.

We are aware of two pointer analysis algorithms for multithreaded programs: an algorithm by Rugina and Rinard for multithreaded programs with structured parbegin/parend concurrency [173], and an intraprocedural algorithm by Corbett [66]. The algorithms are not compositional (they discover the interactions between threads by repeatedly reanalyzing each thread in each new analysis context to reach a fixed point), do not maintain escape information, and do not support the analysis of incomplete programs.

4.7.2 Escape Analysis for Multithreaded Programs

Published escape analysis algorithms for Java programs do not analyze interactions between threads [37, 58, 202, 35]. If an object escapes via a thread object, it is never recaptured. These algorithms are therefore best viewed as sequential program analyses that have been extended to execute correctly but very conservatively in the presence of multithreading. Our analysis takes the next step of analyzing interactions between threads to recapture objects accessed by multiple threads.

Ruf's analysis occupies a point between traditional escape analyses and our multithreaded analysis [172]. His analysis tracks the synchronizations that each thread performs on each object, enabling the compiler to remove synchronizations for objects accessed by multiple threads if only one thread synchronizes on the object. Our analysis goes a step further to remove synchronizations even if multiple threads synchronize on the object. The requirement is that thread start events must temporally separate synchronizations from different threads.

4.7.3 Region-Based Allocation

Region-based allocation has been used in systems for many years. Our comparison focuses on safe versions, which ensure that there are no dangling references to deleted regions. Several researchers have developed type-based systems that support safe region-based allocation [194, 71]. These systems use a flow-insensitive, context-sensitive analysis to correlate the lifetimes of objects with the lifetimes of computations. Although these analyses were designed for sequential programs, it should be straightforward to generalize them to handle multithreaded programs.

Gay and Aiken's system provides an interesting contrast to ours in its overall approach [100]. They provide a safe, flat region-based system that allows arbitrary references between regions. The implementation instruments each store instruction to count references that go between regions. A region can be deleted only when there are no references to its objects from objects in other regions. This dynamic, reference counted approach works equally well for both sequential and multithreaded programs. The system also supports the explicit assignment of objects to regions and allows the programmer to use type annotations to specify that a given reference must stay within the same region. Violations of this constraint generate a run-time error; a static analysis reduces but is not designed to eliminate the possibility of such an error occurring.

Following the Real-Time Java specification, our implementation provides a less flexible system of hierarchically organized regions with an implicit assignment of objects to regions. Because region lifetimes are hierarchically nested, the implementation dynamically counts, for each region, the number of child regions rather than the number of external pointers into each region. Instead of performing counter manipulations at each store, the unoptimized version of our system checks each assignment to ensure that the program never generates a reference that goes down the hierarchy from an ancestor region to a descendant region. Our static analysis eliminates these checks, with the interthread analysis required to successfully optimize multithreaded programs.

4.8 Conclusion

Multithreading is a key program structuring technique, language and system designers have made threads a central part of widely used languages and systems, and multithreaded software is becoming pervasive. This chapter presents an abstraction (parallel interaction graphs) and an algorithm that uses this abstraction to extract precise points-to, escape, and action ordering information for programs that use the standard unstructured form of multithreading provided by modern languages and systems. We have implemented the analysis in the MIT Flex compiler for Java, and used the extracted information to verify that programs correctly use region-based allocation constructs, eliminate dynamic checks associated with the use of regions, and eliminate unnecessary synchronization. Our experimental results show that analyzing the interactions between threads significantly increases the effectiveness of the optimizations for region-based programs, but has little effect for synchronization elimination.

THIS PAGE WAS INTENTIONALLY LEFT BLANK

Chapter 5

Role-Based Exploration of Object-Oriented Programs

5.1 Introduction

This chapter presents a new technique to help developers understand heap referencing properties of object-oriented programs and how the actions of the program affect those properties. Our thesis is that each object's referencing relationships with other objects determine important aspects of its purpose in the computation, and that we can use these referencing relationships to synthesize a set of conceptual object states (we call each state a *role*) that captures these aspects. As the program manipulates objects and changes their referencing relationships, each object transitions through a sequence of roles, with each role capturing the functionality inherent in its current referencing relationships.

We have built two tools that enable a developer to use roles to explore the behavior of object-oriented programs: 1) a dynamic role analysis tool that automatically extracts the different roles that objects play in a given computation and characterizes the effect of program actions on these roles, and 2) a graphical, interactive exploration tool that presents this information in an intuitive form to the developer. By allowing the developer to customize the presentation of this information to show the amount of detail appropriate for the task at hand, these tools support the exploration of both detailed properties within a single data structure and larger properties that span multiple data structures. Our experience using these tools indicates that they can provide substantial insight into the structure, behavior, and key properties of the program and the objects that it manipulates.

5.1.1 Role Separation Criteria

The foundation of our role analysis system is a set of criteria (the *role separation criteria*) that the system uses to separate instances of the same class into different roles. Conceptually, we frame the role separation criteria as a set of predicates that classify objects into roles. Each predicate captures some aspect of the object's referencing relationships. Two objects play the same role if they have the same values for these predicates. Our system supports predicates that capture the following kinds of relationships:

- **Heap Alias Relationships:** The functionality of an object often depends on the objects that refer to it. For example, instances of the `PlainSocketImpl` class acquire input and output capabilities when referred to by a `SocketInputStream` or `SocketOutputStream` object. The role separation criteria capture these distinctions by placing objects with different kinds of heap aliases in different roles. Formally, there is a role separation predicate for each field of each class. An object satisfies the predicate if one such field refers to it.

- **Reference-To Relationships:** The functionality of an object often depends on the objects to which it refers. A Java `Socket` object, for example, does not support communication until its file descriptor field refers to an actual file descriptor object. To capture these distinctions, our role separation criteria place objects in different roles if they differ in which fields contain null values. Formally, there is a predicate for each field of every class. An instance of that class satisfies the predicate if its field is not null.

- **Reachability:** The functionality of an object often depends on the specific data structures in which it participates. For example, a program may maintain two sets of objects: one set that it has completed processing, and another that it has yet to process. To capture such distinctions, our role separation criteria identify the roots of different data structures and place objects with different reachability properties from these roots in different roles. Formally, there is a predicate for each variable that may be a root of a data structure. An object satisfies the predicate if it is reachable from the variable. Additionally, we define a unique *garbage* role for unreachable objects.

- **Identity:** To facilitate navigation, data structures often contain reverse pointers. For example, the objects in a circular doubly-linked list satisfy identity predicates corresponding to the paths `next.prev` and `prev.next`. Formally, there is a role separation predicate for each pair of fields. The predicate is true if the path specified by the two fields exists and leads back to the original object.

- **History:** In some cases, objects may change their conceptual state when a method is invoked on them, but the state change may not be visible in the referencing relationships. For example, the native method `bind` assigns a name to instances of the Java `PlainSocketImpl` class, enabling them to accept connections. But the data structure changes associated with this change are hidden behind the operating system abstraction. To support this kind of conceptual state change, the role separation criteria include part of the method invocation history of each object. Formally, there is a predicate for each parameter of each method. An object satisfies one of these predicates if it was passed as that parameter in some invocation of that method.

5.1.2 Role Subspaces

To allow the developer to customize the role separation criteria, our system supports *role subspaces.* Each role subspace contains a subset of the possible role separation criteria. When operating within a given subspace, the tools coarsen the separation of objects into roles by eliminating any distinctions made only by criteria not in that subspace. Developers may use subspaces in a variety of ways:

- **Focused Subspaces:** As developers explore the behavior of the program, they typically focus on different and changing aspects of the object properties and referencing relationships. By choosing a subspace that excludes irrelevant criteria, the developer can explore relevant properties at an appropriate level of detail while ignoring distracting distinctions that are currently irrelevant.

- **Orthogonal Subspaces:** Developers can factor the role separation criteria into orthogonal subspaces. Each subspace identifies a current role for each object; when combined, the subspaces provide a classification structure in which each object can simultaneously play multiple roles, with each role chosen from a different subspace.

- **Hierarchical Subspaces:** Developers can construct a hierarchy of role subspaces, with child subspaces augmenting parent subspaces with additional role separation criteria. In effect, this approach allows developers to identify an increasingly precise and detailed dynamic classification hierarchy for the roles that objects play during their lifetimes in the computation.

Role subspaces give the developer great flexibility in exploring different perspectives on the behavior of the program. Developers can use subspaces to view changing object states as combinations of roles from different orthogonal role subspaces, as paths through an increasingly detailed classification hierarchy, or as individual points in a constellation of relevant states. Unlike traditional structuring mechanisms such as classes, roles and role subspaces support the evolution of multiple complementary views of the program's behavior, enabling the developer to seamlessly flow through different perspectives as he or she explores different aspects of the program at hand.

5.1.3 Contributions

This chapter makes the following contributions:

- **Role Concept:** It introduces the concept that object referencing relationships and method invocation histories capture important aspects of an object's state, and that these relationships and histories can be used to synthesize a cognitively tractable abstraction for understanding the changing roles that objects play in the computation.

- **Role Separation Criteria:** It presents a set of criteria for classifying instances of the same class into different roles. It also presents an implemented tool that uses these criteria to automatically extract information about the roles that objects play.

- **Role Subspaces:** It shows how developers can use role subspaces to structure their understanding and presentation of the different aspects of the program state. Specifically, the developer can customize the role subspaces to focus the role separation criteria to hide (currently) irrelevant distinctions, to factor the object state into orthogonal components, and to develop object classification hierarchies.

- **Graphical Role Exploration:** It presents a tool that graphically and interactively presents role information. Specifically, this tool presents role transition diagrams, which display the trajectories that objects follow through the space of roles, and role relationship diagrams, which display referencing relationships between objects that play different roles. These diagrams are hyperlinked for easy navigation.

- **Role Exploration Strategy:** It presents a general strategy that we developed to use the tools to explore the behavior of object-oriented programs.

- **Experience:** It presents our experience using our tools on several Java programs. We found that the tools enabled us to quickly discover and understand important properties of these programs.

5.2 Example

We next present a simple example that illustrates how a developer can use our tools to explore the behavior of a web server. We use a version of JhttpServer, a web server written in Java. This program accepts incoming requests for files from web browsers and serves the files back to the web browsers.

The code in the `JhttpServer` class first opens a port and waits for incoming connections. When it receives a connection, it creates a `JhttpWorker` object, passes the Socket controlling the communication to the `JhttpWorker` initializer, and turns control over to the `JhttpWorker` object.

The code in the `JhttpWorker` class first builds input and output streams corresponding to the Socket. It then parses the web browser's request to obtain the requested filename and the http version from the web browser. Next, it processes the request. Finally, it closes the streams and the socket and returns to code in the `JhttpServer` class.

5.2.1 Starting Out

To use our system, the developer first compiles the program using our compiler, then runs the program. The compiler inserts instrumentation code that generates an execution trace. The analysis tool then reads the trace to extract the information and convert it into a form suitable for interactive graphical display. The graphical user interface runs in a web browser with related information linked for easy navigation.

The analysis evaluates the roles of the objects at method boundaries. Our system uses four abstractions to present the observed role information to the developer: 1) role transition diagrams, which present the observed role transitions for instances of a given class, 2) role relationship diagrams, which present referencing relationships between objects from different classes, 3) role definitions, which present the referencing relationships that define each role, and 4) enhanced method interfaces, which show the object referencing properties at invocation and the effect of the method on the roles of the objects that it accesses.

5.2.2 Role Transition Diagrams

Developers typically start exploring the behavior of a program by examining role transition diagrams to get a feel for the different roles that instances of each class play in the computation. In this example, we assume the developer first examines the role transition diagram for the `JhttpWorker` class, which handles client requests. Figure 5-1 presents this diagram.[1] The ellipses represent roles and the arrows represent transitions between roles. Each arrow is labeled with the method that caused the object to take the transition. Solid edges denote the execution of methods that take the `JhttpWorker` as a parameter; dotted edges denote portions of a method or methods that change the roles of `JhttpWorker` objects, but do not take the `JhttpWorker` object as a parameter. The diagram always presents the most deeply nested (in the call graph) method responsible for the role change.

5.2.3 Role Definitions

Role transition diagrams show how objects transition between roles, but provide little information about the roles themselves. Our graphical interface therefore links each role node with its *role definition*, which specifies the properties that all objects playing that role must have. Figure 5-2 presents the role definition for the JhttpWorker with filename role, which is easily accessible by using the mouse to select the role's node in the role transition diagram. This definition specifies that instances of the JhttpWorker with filename role have the class `JhttpWorker`, no heap aliases, no identity relations, and references to heap objects in the fields `httpVersion`, `fileName`, `methodType`, and `client`.

[1]In addition to graphically presenting these diagrams in a web browser, our tool is capable of generating PostScript images of each diagram using the dot tool [86]. Our tool automatically generates initial names for roles and allows the developer to rename the roles. All of the diagrams presented in this chapter were generated automatically from our tool with renaming in some cases for clarification.

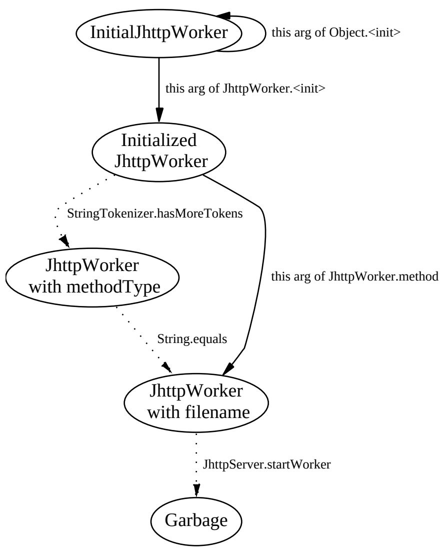

InitialJhttpWorker
this arg of Object.<init>
this arg of JhttpWorker.<init>
Initialized JhttpWorker
StringTokenizer.hasMoreTokens
JhttpWorker with methodType
this arg of JhttpWorker.method
String.equals
JhttpWorker with filename
JhttpServer.startWorker
Garbage

Figure 5-1: Role transition diagram for `JhttpWorker` class

```
Role: JhttpWorker with filename
  Class: JhttpWorker
  Heap aliases: none
  non-null fields: httpVersion, fileName,
                   methodType, client
  identity relations: none
```

Figure 5-2: Sample role definition for `JhttpWorker` class

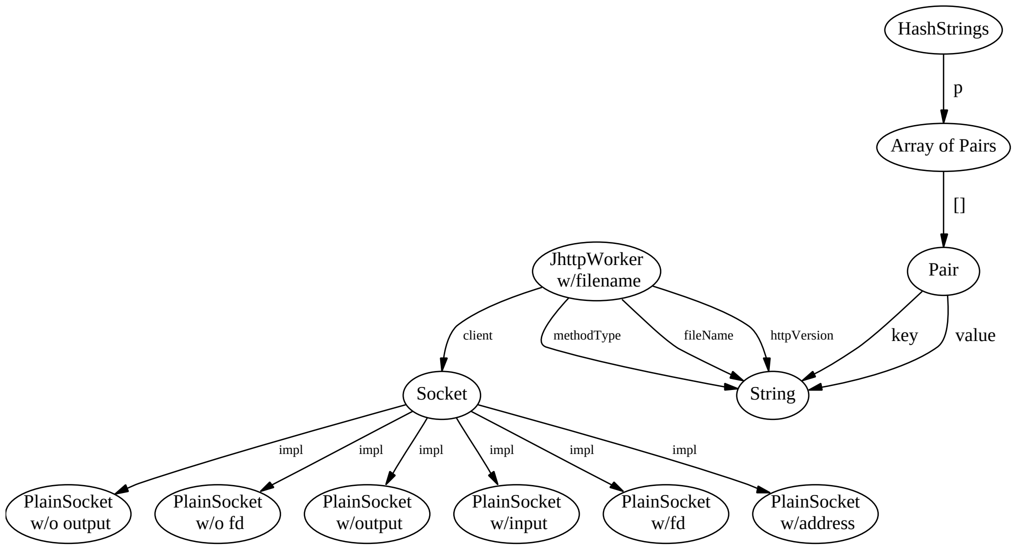


Figure 5-3: Portion of role relationship diagram for JhttpServer

5.2.4 Role Relationship Diagrams

After obtaining an understanding of the roles of important classes, the developer typically moves on to consider relationships between objects of different classes. These relationships are often crucial for understanding the larger data structures that the program manipulates. Role relationship diagrams are the primary tool that developers use to help them understand these relationships. Figure 5-3 presents a portion of the role relationship diagram surrounding one of the roles of the `JhttpWorker` class. The ellipses in this diagram represent roles, and the arrows represent referencing relationships between objects playing those roles.

Note that some of the groups of roles presented in Figure 5-3 correspond to combinations of objects that conceptually act as a single entity. For example, the `HashStrings` object and the underlying array of `Pairs` that it points to implement a map from `String` to `String`. Developers often wish to view a less detailed role relationship diagram that merges the roles for these kinds of combinations.

In many cases, the analysis can automatically recognize these combinations and represent them with a single role node. Figure 5-4 presents the role relationship diagram that the tool produces when the developer turns this option on. Notice that the analysis recognizes the `Socket` object and the `httpVersion` string as being part of the `JhttpWorker` object. Also notice that it recognizes the `Pair` arrays, `Pair` objects, and key strings as being part of the corresponding `HashStrings` object, with the key strings disappearing in the abstracted diagram because they are encapsulated within the `HashStrings` data structure. The analysis allows the developer to choose, for each class, a policy that determines how (and if) the analysis merges roles of that class into larger data structures.

An examination of Figures 5-3 and 5-4 shows that instances of the PlainSocketImpl class play many different roles. To explore these roles, the developer examines the role transition diagram for the PlainSocketImpl class. Figure 5-5 presents this diagram. The diagram contains two disjoint sets of roles, each branching off of the Initial PlainSocket role. This structure indicates that instances of the class have two distinct purposes in the computation. Some instances manage communication over a TCP/IP connection, while others accept incoming connections.

5.2.5 Enhanced Method Interfaces

Finally, our tool can present information about the roles of parameters and the effect of each method on the roles that different objects play. Given a method, our tool presents this information in the form of an enhanced method interface. This interface provides the roles of the parameters at method entry and exit and any read, write, or role transition effects the method may have. Figure 5-6 presents an enhanced method interface for the SocketInputStream initializer. This interface indicates that the SocketInputStream initializer operates on objects that play the roles Initial InputStream and PlainSocket w/fd. When it executes, it changes the roles of these objects to InputStream w/impl and PlainSocket w/input, respectively.

Enhanced method interfaces provide the developer with additional information about the (otherwise implicit) assumptions that the method may make about its parameters and the roles of the objects that it manipulates. This information may help the developer better understand the purpose of the method in the computation and provide guidelines for its successful use in other contexts.

5.2.6 Role Information

In general, roles capture important properties of the objects and provide useful information about how the actions of the program affect those properties.

- **Consistency Properties:** Our analysis can discover program-level data structure consistency properties.
- **Enhanced Method Interfaces:** In many cases, the interface of a method makes assumptions about the referencing relations of its parameters. Our analysis can discover constraints on the roles of parameters of a method and determine the effect of the method on the heap.
- **Multiple Uses:** Code factoring minimizes code duplication by producing general-purpose classes (such as the Java Vector and Hashtable classes) that can be used in a variety of contexts. But this practice obscures the different purposes that different instances of these classes serve in the computation. Our analysis can rediscover these distinctions.
- **Correlated Relationships:** In many cases, groups of objects cooperate to implement a piece of functionality, with the roles of the objects in the group

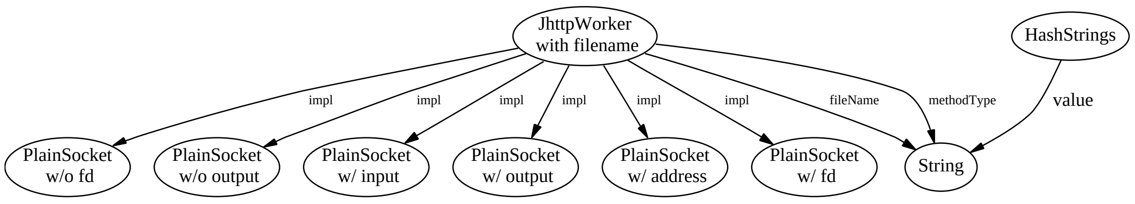


Figure 5-4: Portion of role relationship diagram for JhttpServer after part object abstraction

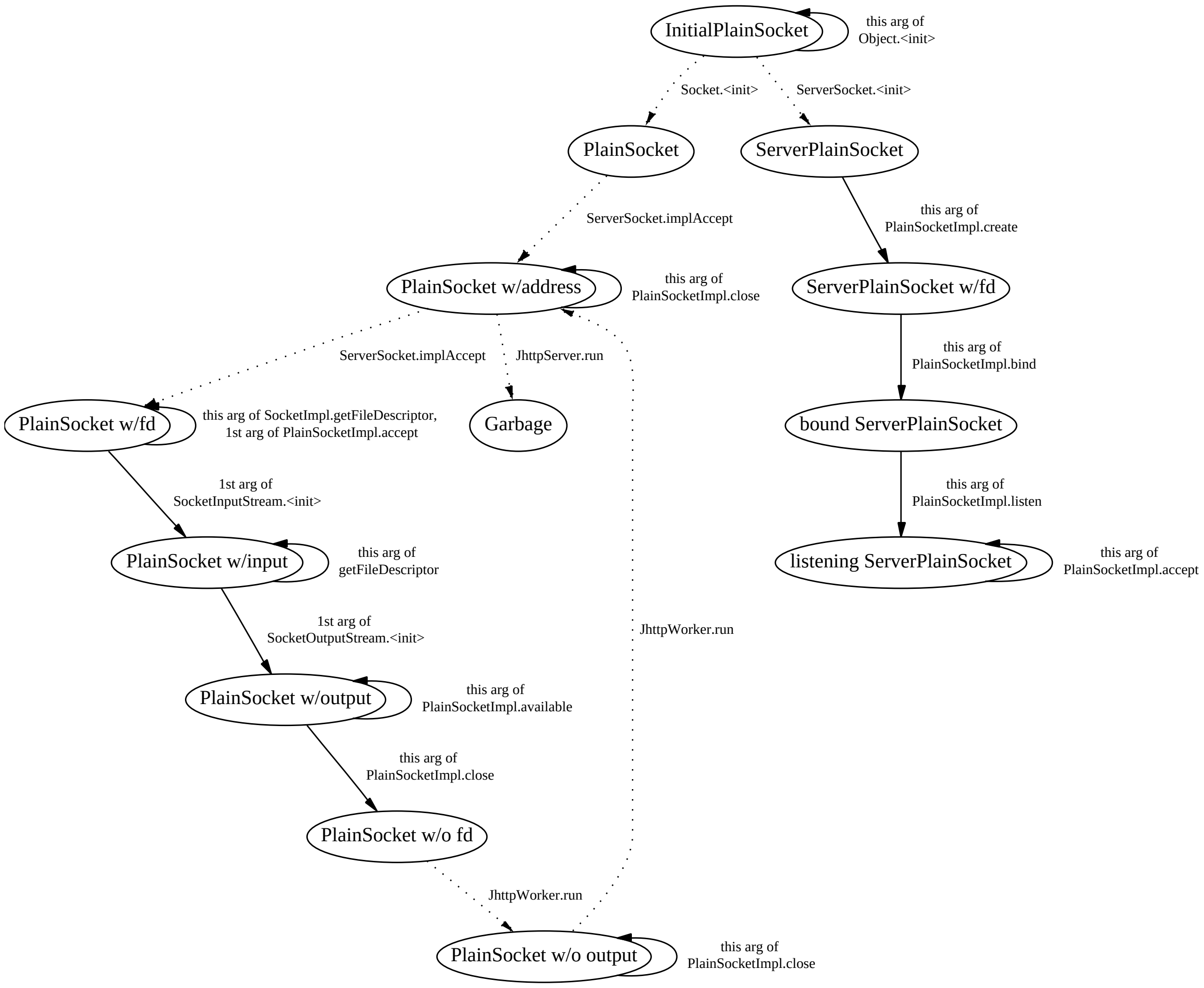

InitialPlainSocket
this arg of Object.<init>
Socket.<init>
ServerSocket.<init>
PlainSocket
ServerPlainSocket
ServerSocket.implAccept
this arg of PlainSocketImpl.create
PlainSocket w/address
this arg of PlainSocketImpl.close
ServerPlainSocket w/fd
ServerSocket.implAccept
JhttpServer.run
this arg of PlainSocketImpl.bind
PlainSocket w/fd
this arg of SocketImpl.getFileDescriptor, 1st arg of PlainSocketImpl.accept
Garbage
bound ServerPlainSocket
1st arg of SocketInputStream.<init>
this arg of PlainSocketImpl.listen
PlainSocket w/input
this arg of getFileDescriptor
listening ServerPlainSocket
this arg of PlainSocketImpl.accept
1st arg of SocketOutputStream.<init>
JhttpWorker.run
PlainSocket w/output
this arg of PlainSocketImpl.available
this arg of PlainSocketImpl.close
PlainSocket w/o fd
JhttpWorker.run
PlainSocket w/o output
this arg of PlainSocketImpl.close

Figure 5-5: Role transition diagram for the `PlainSocketImpl` class

```
Method: SocketInputStream.<init>(this,plainsocket)
 Call Context: {
  this: Initial InputStream -> InputStream w/impl,
  plainsocket: PlainSocket w/fd ->
    PlainSocket w/input }
 Write Effects:
  this.impl=plainsocket
  this.temp=NEW
  this.fd=plainsocket.fd
 Read Effects:
  plainsocket
  NEW
  plainsocket.fd
 Role Transition Effects:
  plainsocket: PlainSocket w/fd -> PlainSocket
                                   w/input
  this: Initial InputStream -> InputStream w/fd
  this: InputStream w/fd -> InputStream w/impl
```

Figure 5-6: Enhanced Method Interface for `SocketInputStream` initializer

changing together over the course of the computation. Our analysis can discover these correlated state changes.

5.3 Dynamic Analysis

We implemented the dynamic analysis as several components. The first component uses the MIT FLEX compiler [2] to instrument Java programs to generate execution traces. Because this component operates on Java bytecodes, our system does not require source code. The instrumented program assigns unique identifiers to every object and reports relevant heap and pointer operations in the execution trace. The second component uses the trace to reconstruct the heap. As part of this computation, it also calculates reachability information and records the effect of each method's execution on the roles of the objects that it manipulates.

5.3.1 Predicate Evaluation

The dynamic analysis uses the information it extracts from the trace to apply the role separation criteria as follows:

- **Heap Aliases:** In addition to reconstructing the heap, the analysis also maintains a set of *inverse references*. There is one inverse reference for each reference

[2]Available at www.flexc.lcs.mit.edu.

in the original heap. For each reference to a target object, the inverse reference enables the dynamic analysis to quickly find the source of the reference and the field containing the reference. To compute the heap alias predicates for a given object, the analysis examines the inverse references for that object.

- **Reference-To:** The reconstructed heap contains all of the references from the original program, enabling the analysis to quickly compute all of the reference-to predicates for a given object by examining its list of references.

- **Identity:** To compute the identity predicates for a given object, the analysis traces all paths of length two from the object to find paths that lead back to the object.

- **Reachability:** There are two key issues in computing the reachability information: using an efficient incremental reachability algorithm and choosing the correct set of variables to include in the role separation criteria. Whenever the program changes a reference, the incremental reachability algorithm finds the object whose reachability properties may have changed, and then incrementally propagates the reachability changes through the reconstructed heap.

  To avoid undesirable separation caused by an inappropriate inclusion of temporary variables into the role separation criteria, our implemented system uses two rules to identify variables that are the roots of data structures. If an object o is reachable from variables x and y that point to objects o_x and o_y respectively, and o_x is reachable from y but o_y is not reachable from x, then we exclude x from the role separation criteria. Alternatively, if o_x is reachable from y, o_y is reachable from x, and the reference y was created before the reference x, we exclude x from the criteria.

  These rules keep temporary references used for traversing heap structures from becoming part of the role definitions, but allow long term references to the roots of data structures to be incorporated into role definitions. These rules also have the property that if an object is included in two disjoint data structures with different roots, then the object's role will reflect this double inclusion.

- **Method Invocation History:** Whenever an object is passed as a parameter to a method, the analysis records the invocation as part of the object's method invocation history. This record is then used to evaluate method invocation history predicates when assigning future roles to the object.

- **Array Roles:** We treat arrays as objects with a special `[]` field, which points to the elements of the array. Additionally, we generalize the treatment of reference-to relations to allow roles to specify the classes and the corresponding number (up to some bound) of the array's elements.

By default, the analyzer evaluates these predicates at every method entry and exit point. We allow the developer to coarsen this granularity by declaring methods *atomic*, in which case the analysis attributes all role transitions that occur inside the method to the method itself. This is implemented by not checking for role transitions until the atomic method returns. This mechanism hides temporary or irrelevant role transitions that occur inside the method. This feature is most useful for simplifying role transition diagrams. In particular, many programs have a complicated process for initializing objects. Once we use the role transition diagram to understand this process, we often find it useful to abstract the entire initialization process as atomically generating a fully initialized object.

5.3.2 Multiple Object Data Structures

A single data structure often contains many component objects. Java `HashMap` objects, for example, use an array of linked lists to implement a single map. To enable the developer to view such composite data structures as a single entity, our dynamic analysis supports operations that merge multiple objects into a single entity. Specifically, the dynamic analysis can optionally recognize any object playing a given role (such roles are called *part roles*) as conceptually part of the object that refers to it. The user interface will then merge all of the role information from the part role into the role of the object that refers to it.

Depending on the task at hand, different levels of abstraction may be useful to the developer. On a per class basis, the developer can specify whether to merge one object's role into another object's role. The analysis provides four different policies: never merge, always merge, merge only if one heap reference to the object ever exists, and merge only if one heap reference at a time exists to the object. The analysis implements these policies using a two pass strategy: one pass identifies concrete objects that meet the merging criterion, and another assigns the selected objects part roles. The analysis requires that any cycles in the heap include at least one object that does not have a part role.

5.3.3 Method Effect Inference

For each method execution, the dynamic analysis records the reads, writes, and role transitions that the execution performs. Each method effect summary uses regular expressions to identify paths to the accessed or affected objects. These paths are identified relative to the method parameters or global variables and specify edges in the heap that existed when the method was invoked. Method effect inference therefore has two steps: detecting concrete paths with respect to the heap at procedure invocation and summarizing these paths into regular expressions.

To detect concrete paths, we keep a path table for each method invocation. This table contains the concrete path, in terms of the heap that existed when the method was invoked, to all objects that the execution of the method may affect. At method invocation, our analysis records the objects to which the parameters and the global variables point. Whenever the execution retrieves a reference to an object or changes an object's reachability information, the analysis records a path to that object in the path table. If the execution creates a new object, we add a special NEW token to the path table; this token represents the path to that object.

We obtain the regular expressions in the method effect summary by applying a set of rewrite rules to the extracted concrete paths. Figure 5-7 presents the current set of rewrite rules. Given a concrete path $f_1.f_2...f_n$, we apply the rewrite rules to the tuple $\langle \epsilon, f_1.f_2...f_n \rangle$ to obtain a final tuple $\langle Q, \epsilon \rangle$, where Q is the regular expression that represents the path. We present the rewrite rules in the order in which they are applied. We use the notation that $\kappa(f)$ denotes the class in which the field f is declared as an instance variable, and $\tau(f)$ is the declared type of the field f.

Rules 1 and 2 simplify intermediate expressions generated during the rewrite process. Rules 3 and 4 generalize concrete paths involving similar fields such as paths through a binary tree. Rules 5 and 6 generalize repeated sequences in concrete paths. The goal is to capture paths generated in loops or recursive methods and ensure that path expressions are not overly specialized to any particular execution.

1. $\langle Q.(q_1...(e_1 \mid f \mid e_2 \mid f \mid e_3)...q_n)^*, Q' \rangle \Rightarrow$
   $\langle Q.(q_1...(e_1 \mid f \mid e_2 \mid e_3)...q_n)^*, Q' \rangle$

2. $\langle Q.(q_1...(e_1 \mid f \mid e_2 \mid f \mid e_3)^*...q_n)^*, Q' \rangle \Rightarrow$
   $\langle Q.(q_1...(e_1 \mid f \mid e_2 \mid e_3)^*...q_n)^*, Q' \rangle$

3. $\langle Q.(f_1), f_2.Q' \rangle \Rightarrow \langle Q.(f_1 \mid f_2)^*, Q' \rangle$
   if $\kappa(f_1) = \kappa(f_2)$ and $\tau(f_1) = \tau(f_2)$

4. $\langle Q.(f_0 \mid ... \mid f_n)^*, f'.Q' \rangle \Rightarrow \langle Q.(f_0 \mid ... \mid f_n \mid f')^*, Q' \rangle$
   if $\kappa(f_n) = \kappa(f')$ and $\tau(f_n) = \tau(f')$

5. $\langle Q.q_1...q_n.q'_1...q'_n, Q' \rangle \Rightarrow \langle Q.(q_1 \oplus q'_1...q_n \oplus q'_n)^*, Q' \rangle$
   if $\forall i, 1 \le i \le n, q_i \equiv q'_i$, where $q \equiv q'$ if
   (a) $q = (f_1 \mid ... \mid f_j), q' = (f'_1 \mid ... \mid f'_k)$,
   $\kappa(f_1) = \kappa(f'_1)$ and $\tau(f_1) = \tau(f'_1)$, or
   (b) $q = (f_1 \mid ... \mid f_j)^*, q' = (f'_1 \mid ... \mid f'_k)^*$,
   $\kappa(f_1) = \kappa(f'_1)$ and $\tau(f_1) = \tau(f'_1)$.
   $(f_1 \mid ... \mid f_j) \oplus (f'_1 \mid ... \mid f'_k) = (f_1 \mid ... \mid f_j \mid f'_1 \mid ... \mid f'_k)$
   $(f_1 \mid ... \mid f_j)^* \oplus (f'_1 \mid ... \mid f'_k)^* =$
   $(f_1 \mid ... \mid f_j \mid f'_1 \mid ... \mid f'_k)^*$

6. $\langle Q.(q_1...q_n)^*.q'_1...q'_n, Q' \rangle \Rightarrow \langle Q.(q_1 \oplus q'_1...q_n \oplus q'_n)^*, Q' \rangle$
   if $\forall i, 1 \le i \le n, (q_i \equiv q'_i)$.

7. $\langle Q, f.Q' \rangle \Rightarrow \langle Q.(f), Q' \rangle$

Figure 5-7: Rewrite rules for paths

For read or role transition effects, we record the starting point and regular expression for the path to the object. For write effects, we give the starting points for both objects and the regular expressions for the paths. Valid starting points are method parameters and global variables. We denote effects for objects created in a procedure using the NEW token. We denote writing a null pointer to an object's field using the NULL token.

5.3.4 Role Subspaces

Our tool allows the developer to define multiple role subspaces and modify the role separation criteria for each subspace as follows:

- **Fields:** The developer can specify fields to ignore for the purpose of assigning roles. The analysis will show these fields in the role relationship diagram, but the references in these fields will not affect the roles assigned to the objects.
- **Methods:** The developer can specify which methods and which parameters to include in the role separation criteria.
- **Reachability:** The developer can specify variables to include or to exclude from the reachability-based role separation criteria.
- **Classes:** The developer can collapse all objects of a given class into a single role.

In practice, we have found role subspaces both useful and usable — useful because they enabled us to isolate the important aspects of relevant parts of the system while eliminating irrelevant and distracting detail in other parts, and usable because we were usually able to obtain a satisfactory role subspace with just a small number of changes to the default criteria.

5.4 User Interface

The user interface presents four kinds of web pages: class pages, role pages, method pages, and the role relationship page. Each class page presents the role transition diagram for the class. From the class page, the developer can click on the nodes and edges in the role transition diagram to see the corresponding role and method pages for the selected node or edge. Each role page presents a role definition, displaying related roles and classes and enabling the developer to select these related roles and classes to bring up the appropriate role or class page. Each method page shows the developer which methods called the given method and allows the developer to configure method-specific abstraction policies. The role relationship page presents the role relationship diagram. From this diagram, the developer can select a role node to see the appropriate role definition page.

The user interface allows the developer to create and manipulate multiple role subspaces. The developer can create a new role subspace by selecting a set of predicates to determine the role separation criteria, then combine subspaces to define views. Views with a single subspace use the role separation criteria from that subspace. Views with multiple subspaces use a cross product operator to combine the roles from the different subspaces, with the final set of roles isomorphic to those obtained by taking the union of the role separation criteria from all of the subspaces. Within a view, the developer can identify additional role subspaces to be used for labeling purposes. These role subspaces do not affect the separation of objects into roles, but rather label each role in the view with the roles that objects playing those roles have in these additional labeling subspaces.

5.5 Exploration Strategy

As we used the tool, we developed the following strategy for exploring the behavior of a new program. We believe this strategy is useful for structuring the process of using the tool, and that most developers will use some variant of this strategy.

When we started using the tool on a new program, we first recompiled the program with our instrumentation package, and then ran the program to obtain an execution trace. We then used our graphical tool to browse the role transition diagrams for each of the classes, looking for interesting initialization sequences, splits in the role transition diagram indicating different uses for objects of the class, and transition sequences indicating potential changes in the purpose of instances of the class in the computation.

During this activity, we were interested in obtaining a broad overview of the actions of the program. We therefore often found opportunities to appropriately simplify the role transition diagrams, typically by creating a role subspace to hide irrelevant detail, by declaring initializing methods atomic, or by utilizing the multiple object abstraction feature. Occasionally, we found opportunities to include aspects of the method invocation history into the role separation criteria. We found that our default policy for merging multiple object data structures into a single data structure for role presentation purposes worked well during this phase of the exploration process.

Once we had created role subspaces revealing roles at an appropriate granularity, we then browsed the enhanced method interfaces to discover important constraints on the roles of the objects passed as parameters to the method. This information enabled us to better understand the correlation between the actions of the method and the role transitions, helping us to isolate the regions of the program that performed important modifications, such as insertions or removals from collections. It also helped us understand the (otherwise implicit) assumptions that each method made about the states of its parameters. We found this information useful in understanding the program; we expect maintainers to find it invaluable.

We next observed the role relationship diagram. This diagram helped us to better understand the relationships between classes that work together to implement a given piece of functionality. In general, we found that the complete role relationship diagram presented too much information for us to use it effectively. We therefore adopted a strategy in which we identified a starting class of interest, then viewed the region surrounding the roles of that class. We found that this strategy enabled us to quickly and effectively find the information we needed in the role relationship diagram.

Finally, we sometimes decided to explore several roles in more detail. We often returned to the role transition diagram and created a customized role subspace to expose more detail for the current class but less detail for less relevant classes. In effect, this activity enabled us to easily adapt the system to view the program from a more specialized perspective. Given our experience using this feature of our role analysis tool, we believe that this ability will prove valuable for any program understanding tool.

5.6 Experience

We next discuss our experience using our role analysis tool to explore the behavior of several Java programs. We report our experience for several programs: Jess, an expert system shell in the SpecJVM benchmark suite; Direct-To, a Java version of an air-traffic control tool; Tagger, a text formatting program; Treeadd, a tree manipulation benchmark in the J. Olden benchmark suite [3]; and Em3d, a scientific computation in the J. Olden benchmark suite.

5.6.1 Jess

Jess first builds a network of nodes, then performs a computation over this network. While the network contains many different kinds of nodes, all of the nodes exhibit a similar construction and use pattern. Consider, for example, instances of the `Node1TELN` class. Figure 5-8 presents the role transition diagram for objects of this class. An examination of this diagram and the linked role definitions shows that during the construction of the network, the program represents the edges between nodes using a resizable vector of references to `Successor` objects, each of which is a wrapper around a node object. The `succ` field refers to this vector. When the network is complete, the program constructs a less flexible but more efficient representation in which each node contains a fixed-size array of references to other nodes; the `_succ` field refers to this array. This change occurs when the program invokes the `freeze` method on the node. All of the nodes in the program exhibit this construction pattern.

The generated method annotations provide information about the assumptions that several key methods make about the roles of their parameters. Specifically, these annotations show that the program invokes the `CallNode` method (this method implements the primary computation on the network) on a node only after the `freeze` method has converted the representation of the edges associated with the node to the more efficient form.

The role definitions also provide information about network's structure, specifically that all of the nodes in the network have either one or two incoming edges. Each fully constructed instance of the `Node1TELN`, `Node1TECT`, `Node1TEQ`, `NodeTerm`,

[3]Available at www-ali.cs.umass.edu/~cahoon.

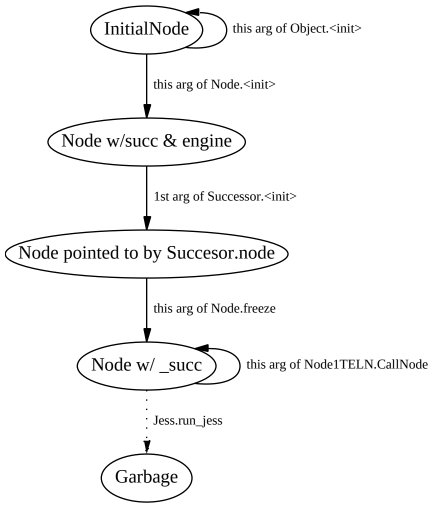

InitialNode
this arg of Object.<init>
this arg of Node.<init>
Node w/succ & engine
1st arg of Successor.<init>
Node pointed to by Succesor.node
this arg of Node.freeze
Node w/ _succ
this arg of Node1TELN.CallNode
Jess.run_jess
Garbage

Figure 5-8: Role transition diagram for the `Node1TELN` class

or `Node1TMF` class has exactly one `Successor` object that refers to it, indicating that these kinds of nodes all have exactly one incoming edge. Each fully constructed instance of the `Node2` class, on the other hand, has exactly two references from `Successor` objects, indicating that `Node2` nodes have exactly two incoming edges.

5.6.2 Direct-To

Direct-To is a prototype Java implementation of a component of the Center-Tracon Automation System (CTAS) [128]. The tool helps air-traffic controllers streamline flight paths by eliminating intermediate points; the key constraint is that these changes should not cause new conflicts, which occur when aircraft pass too close to each other.

We first discuss our experience with the `Flight` class, which represents flights in progress. Each `Flight` object contains references to other objects, such as `FlightPlan` objects and `Route` objects, that are part of its state. Our analysis recognized these other objects as part of the corresponding `Flight` object's state, and merged all of these objects into a single multiple object data structure.

Roles helped us understand the initialization sequence and subsequent usage pattern of Flight objects. An initialized `Flight` object has been inserted into the flight list; various fields of the object refer to the objects that implement the flight's identifier, type, aircraft type, and flight plan. Once initialized, the flight is ready to participate in the main computation of the program, which repeatedly acquires a radar track for the flight and uses the track and the flight plan to compute a projected trajectory. The initialization sequence is clearly visible in the role transition diagram, which shows a linear sequence of role transitions as the flight object acquires references to its part objects and is inserted into the list of flights. The acquisition and computation of the tracks and trajectories also show up as transitions in this diagram.

Roles also enabled us to untangle the different ways in which the program uses instances of the `Point4d` class. Specifically, the program uses instances of this class to represent aircraft tracks, trajectories, and velocities. The role transition diagram makes these different uses obvious: each use corresponds to a different region of roles in the diagram. No transitions exist between these different regions, indicating that the program uses the corresponding objects for disjoint purposes.

5.6.3 Tagger

Tagger is a document layout tool written by Daniel Jackson. It processes a stream of text interspersed with tokens that identify when conceptual components such as paragraphs begin and end. Tagger works by first attaching action objects to each token, and then processing the text and tokens in order. Whenever it encounters a token, it executes the attached action.

It turns out that there are dependences between the operations of the program and the roles of the actions and tokens. For example, one of the tokens causes the output of the following paragraph to be suppressed. Tagger implements this suppression action with pairs of matched suppress/unsuppress actions. When the suppress action executes, it places an unsuppress action at the end of the paragraph, ensuring that only one paragraph will be suppressed. These actions are reflected in role transitions as follows. When the program binds the suppress action to a token, the action takes a transition because of the reference from the token. When the suppress action executes, it binds the corresponding unsuppress action to the token at the end of the paragraph, causing the unsuppress action to take a transition to a new state. Roles therefore enabled us to discover an interesting correlation between the execution of the suppress action and data structure modifications required to undo the action later. We were also able to observe a role-dependent interface — the method that executes actions always executes actions that are bound to tokens.

5.6.4 Treeadd

Treeadd builds a tree of `TreeNode` objects; each such object has an integer value field. It then calculates the sum of the values of the nodes. The role analysis tool extracted some interesting properties of the data structure and gave us insight into the behavior of the parts of the program that construct and use the tree.

Figure 5-9 presents the region of the role relationship diagram that contains the roles of `TreeNode` objects. By examining this diagram and the linked role definitions, we were able to determine that the `TreeNode` objects did in fact comprise a tree — the roles corresponding to the root of the tree have no references from `left` or `right` fields of other `TreeNode` objects, and all other `TreeNode` roles have exactly one reference from the `left` or `right` field of another `TreeNode`.

Figure 5-10 presents the role transition diagram for `TreeNode` objects. This diagram, in combination with the linked role definitions, clearly shows a bottom-up

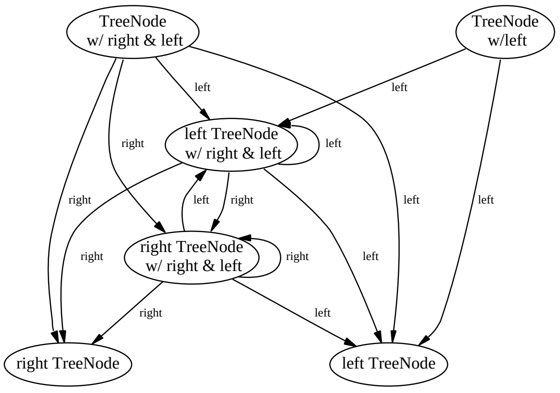
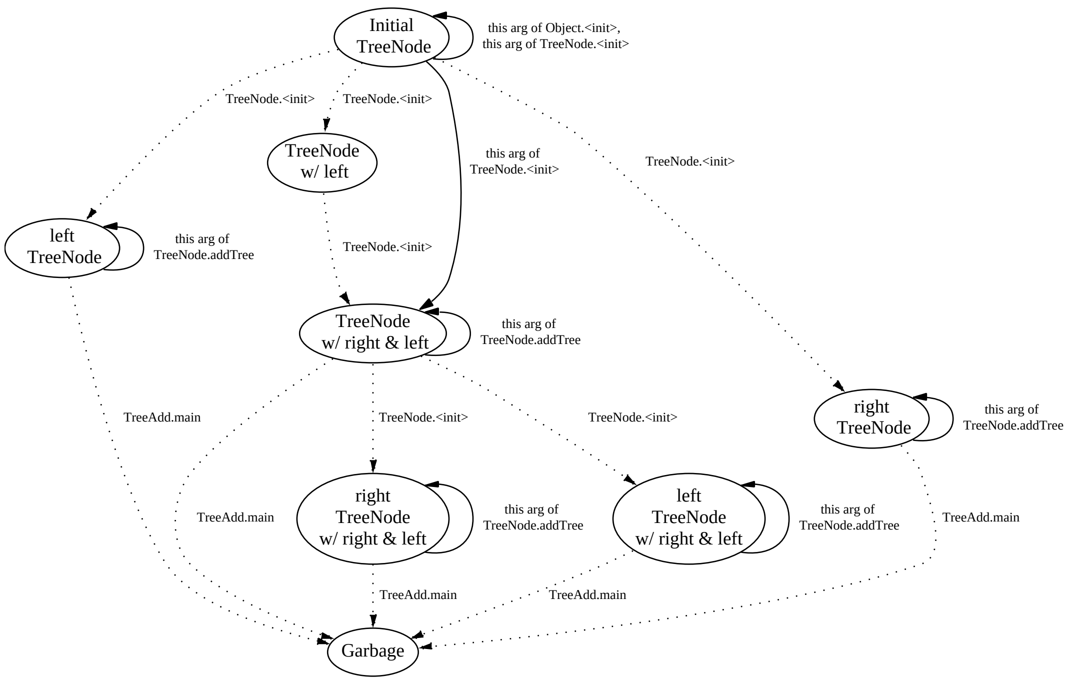


Figure 5-9: Role relationship diagram for the `TreeNode` class


Initial TreeNode
this arg of Object.<init>, this arg of TreeNode.<init>
TreeNode.<init>
TreeNode w/ left
this arg of TreeNode.<init>
left TreeNode
this arg of TreeNode.addTree
TreeNode w/ right & left
right TreeNode
TreeAdd.main
right TreeNode w/ right & left
left TreeNode w/ right & left
Garbage
Figure 5-10: Role transition diagram for the `TreeNode` class

initialization sequence in which each TreeNode acquires a left child and a right child, then a reference from the right or left field of its parent. Alternative initialization sequences produce TreeNode objects with no children. Note that the automatically generated role names in this figure are intended to help the developer understand the referencing relationships that define each role. The role name Right TreeNode w/right & left, for example, indicates that objects playing the role have 1) a reference from the right field of an object, and 2) non-null right and left fields. The role name TreeNode w/left indicates that an object playing this role has a non-null left field.

5.6.5 Em3d

Em3d simulates the propagation of electromagnetic waves through objects in three dimensions. It uses enumerators extensively in two phases of the computation. The first phase builds a graph that models the electric and magnetic fields; the second phase traverses the graph to simulate the propagation of these fields. The role transition diagram for the enumerator objects contains roles corresponding to an initialized enumerator, an enumerator with remaining elements, and an enumerator with no remaining elements. As expected, the program never invokes the next method on an enumerator object that has no remaining elements, enabling the developer to verify that the program uses enumerator objects in a standard way.

5.6.6 Utility of Roles

In general, roles helped us to discover key data structure properties and understand how the program initialized and manipulated objects and data structures. The combination of the role relationship diagram and linked role definitions typically provided the most useful information about data structure properties. Examples of these properties include the referencing properties of TreeNode objects in the Treeadd benchmark and the correspondence between Successor nodes and network nodes in Jess.

The role transition diagram typically provided the most useful information about object initialization sequences and usage patterns. Examples of object initialization sequences include the initialization of Flight objects in the Direct-to benchmark and of TreeNode objects in the Treeadd benchmark. Jess provides an interesting example of a conceptual phase transition in a data structure — the program uses a more flexible but less efficient data structure during a construction phase, then replaces this data structure with a more efficient frozen version for a subsequent computation phase. The Point4d class in Direct-to provides a good example of how a program can use instances of a single class for several different purposes in the computation. In all of these cases, the role analysis enabled us to quickly understand the underlying initialization sequences or usage patterns.

Finally, we found that the information about the roles of method parameters helped us to understand the otherwise implicit expectations that methods have about the states of their parameters and the effects of methods on these states. Examples of methods with important expectations or effects include the `freeze` and `CallNode` methods in Jess and the `next` method in Em3d. In general, we expect the role analysis tool to be useful in the software development process in the following ways:

- **Program Understanding:** Developers have to understand programs to modify or reuse them. In object-oriented languages, understanding heap allocated data structures is key to understanding the program. Roles help developers discover key data structure invariants and understand how programs initialize and manipulate these data structures, thus aiding program comprehension.

- **Maintenance:** To safely modify programs, developers need to understand the data structures these programs build, the referencing relations methods assume, and the effects of methods on these data structures. We expect that the diagrams and enhanced method interfaces that our tool generates will prove useful for this purpose.

- **Verifying Expected Behavior:** Developers can use our tool as a debugging aid. Developers write programs with certain invariants about heap structures in mind. If the role relationships our tool discovers are inconsistent with these invariants, the developer knows that a bug exists. Finally, the enhanced method interfaces and role transition diagrams can help the developer quickly isolate the bug.

- **Documentation:** Developers often need to document high-level properties of the program. Roles may provide an effective documentation mechanism, because they come with a set of appealing interactive graphical representations, because they can often capture key properties of the program in a concise, cognitively tractable representation, and because (at least for the roles that our analysis tool discovers) they are guaranteed to faithfully reflect some of the behaviors of the program. Role subspaces may prove to be especially useful in presenting focused, orthogonal, or hierarchical perspectives on the purposes of the objects in the program.

- **Design:** High-level design formalisms often focus on the conceptual states of objects and the relationships between objects in these states. Our role analysis can extract information that is often similar to this design information, helping the developer to establish the connection between the design and the behavior of the program. Furthermore, the role abstraction suggests several concrete ways of realizing high-level design patterns in the code. As developers become used to working with roles, they may very well adopt role-inspired coding styles that facilitate the verification of a guaranteed connection between the high-level design and its realization in the program.

5.7 Related Work

We survey related work in three fields: design formalisms that involve the concept of abstract object states, program understanding tools that focus on properties of the objects that programs manipulate, and static analyses for automatically discovering or verifying properties of linked data structures.

5.7.1 Design Formalisms

Early design formalisms identified changes in abstract object or component states as an important aspect of the design of the program [166]. Our tool also focuses on abstract state changes as a key aspect, but uses the role separation criteria to automatically synthesize a set of abstract object states rather than relying on the developer to specify the abstract state space explicitly.

Object models enable a developer to describe relationships between objects, both at a conceptual level and as realized in programs. Object modeling languages such as UML [161] and Alloy [127] can describe the different states that objects can be in, the constraints that these states satisfy, and the transitions between these states. One can view our role analysis tool as a way of automatically extracting an object model that captures the important aspects of the objects that the program manipulates. In this sense our tool establishes a connection between the abstract concepts in the object model and the concrete realization of those concepts in the objects that the program manipulates.

The concept of objects playing different roles in the computation while maintaining their identity often arises in the conceptual design of systems [94], and researchers have proposed several methodologies for realizing these roles in the program [94, 91, 130]. Our role analysis tool can recognize many of the design patterns used to implement these roles, and may therefore help developers establish a connection between an existing conceptual system design and its realization in the program. Conversely, our role separation criteria may also suggest alternate ways to implement conceptual roles. In particular, previously proposed methodologies tend to focus on ways to tag objects with (potentially redundant) information indicating their roles, while the role separation criteria identify data structure membership (which may not be directly observable in the state of the object itself) as an important property that helps to determine the roles that the object plays.

5.7.2 Program Understanding Tools

Daikon [88] extracts likely algebraic invariants from information gathered during the program's execution. For example, Daikon can infer invariants such as "$y = 2x$". Daikon handles heap structures in a limited fashion by linearizing them into arrays under some specific conditions [89]. Our work differs in that we handle heap structures in a much more general fashion and focus on referencing relationships as opposed to algebraic invariants.

Womble [129] and Chava [137] both use a static analysis to automatically extract object models for Java programs. Both tools use information from the class and field declarations; Womble also uses a set of heuristics to generate conjectures regarding associations between classes, field multiplicities, and mutability.

Unlike our role analysis tool, Womble and Chava do not support the concept of an object that changes state during the execution of the program. They instead statically group all instances of the same class into the same category of objects in the object model, ignoring any conceptual state changes that may occur because of method invocations, changes to the object referencing relationships, or reachability changes.

5.7.3 Verifying Data Structure Properties

The analysis presented in this chapter extracts role information for a single execution of the program. While it would be straightforward to combine information from multiple executions, the tool is not designed to extract or verify role information that is guaranteed to fully characterize all executions.

Statically extracting or verifying the detailed object referencing properties that roles characterize is clearly beyond the capabilities of standard pointer analysis algorithms. Researchers in our group have, however, been able to leverage techniques from precise shape analysis algorithms to develop an augmented type system and analysis algorithm that is capable of verifying that all executions of a program respect a given set of role declarations [140]. In this context, our dynamic tool could generate candidate role declarations for existing programs. Such a candidate generation system would have to be designed carefully — we expect the dynamic role analysis to be capable of extracting properties that are beyond the verification capabilities of the static role analysis.

5.8 Conclusion

We believe that roles are a valuable abstraction for helping developers to understand the objects and data structures that programs manipulate. We have implemented a dynamic role analysis tool and a flexible interactive graphical user interface that helps developers navigate the information that the analysis produces. Our experience with several Java applications indicates that our tools can help developers discover important object initialization sequences, object usage patterns, data structure invariants, and constraints on the states and referencing relationships of method parameters. Other potential applications include documenting high-level properties of the program (and especially properties that involve orthogonal or hierarchical object and data structure classification structures), discovering correlated state changes between objects that participate in the same data structure, providing specifications for a static role analysis algorithm, verifying or refuting a debugger's hypotheses about important data structure invariants, and providing a foundation for establishing a guaranteed connection between the high-level design and its realization in the program.

Chapter 6

Role Analysis

Types capture important properties of the objects that programs manipulate, increasing both the safety and readability of the program. Traditional type systems capture properties (such as the format of data items stored in the fields of the object) that are invariant over the lifetime of the object. But in many cases, properties that do change are as important as properties that do not. Recognizing the benefit of capturing these changes, researchers have developed systems in which the type of the object changes as the values stored in its fields change or as the program invokes operations on the object [188, 187, 74, 206, 207, 54, 109, 84]. These systems integrate the concept of changing object states into the type system.

The fundamental idea in this work is that the state of each object also depends on the data structures in which it participates. Our type system therefore captures the referencing relationships that determine this data structure participation. As objects move between data structures, their types change to reflect their changing relationships with other objects. Our system uses *roles* to formalize the concept of a type that depends on the referencing relationships. Each role declaration provides complete aliasing information for each object that plays that role—in addition to specifying roles for the fields of the object, the role declaration also identifies the complete set of references in the heap that refer to the object. In this way roles generalize linear type systems [199, 29, 136] by allowing multiple aliases to be statically tracked, and extend alias types [183, 200] with the ability to specify roles of objects that are the source of aliases.

This approach attacks a key difficulty associated with state-based type systems: the need to ensure that any state change performed using one alias is correctly reflected in the declared types of the other aliases. Because each object's role identifies all of its heap aliases, the analysis can verify the correctness of the role information at all remaining or new heap aliases after an operation changes the referencing relationships.

Roles capture important object and data structure properties, improving both the safety and transparency of the program. For example, roles allow the programmer to express data structure consistency properties (with the properties verified by the role analysis), to improve the precision of procedure interface specifications (by allowing the programmer to specify the role of each parameter), to express precise referenc-

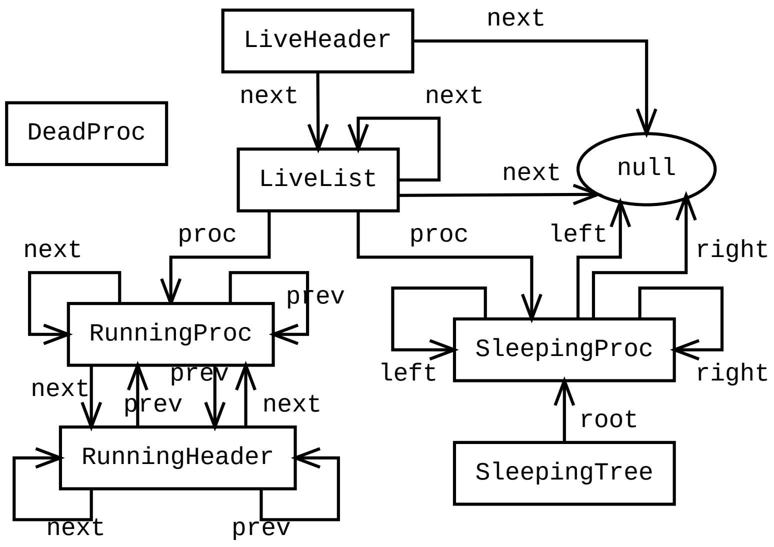


Figure 6-1: Role Reference Diagram for a Scheduler

ing and interaction behaviors between objects (by specifying verified roles for object fields and aliases), and to express constraints on the coordinated movements of objects between data structures (by using the aliasing information in role definitions to identify legal data structure membership combinations). Roles may also aid program optimization by providing precise aliasing information.

6.1 Overview of Roles

Figure 6-1 presents a *role reference diagram* for a process scheduler. Each box in the diagram denotes a disjoint set of objects of a given role. The labelled arrows between boxes indicate possible references between the objects in each set. As the diagram indicates, the scheduler maintains a list of live processes. A live process can be either running or sleeping. The running processes form a doubly-linked list, while sleeping processes form a binary tree. Both kinds of processes have `proc` references from the live list nodes `LiveList`. Header objects `RunningHeader` and `SleepingTree` simplify operations on the data structures that store the process objects.

As Figure 6-1 shows, data structure participation determines the conceptual state of each object. In our example, processes that participate in the sleeping process tree data structure are classified as sleeping processes, while processes that participate in the running process list data structure are classified as running processes. Moreover, movements between data structures correspond to conceptual state changes—when a process stops sleeping and starts running, it moves from the sleeping process tree to the running process list.

6.1.1 Role Definitions

Figure 6-2 presents the role definitions for the objects in our example.[1] Each role definition specifies the constraints that an object must satisfy to play the role. Field constraints specify the roles of the objects to which the fields refer, while slot constraints identify the number and kind of aliases of the object.

Role definitions may also contain two additional kinds of constraints: identity constraints, which specify paths that lead back to the object, and acyclicity constraints, which specify paths with no cycles. In our example, the identity constraint `next.prev` in the `RunningProc` role specifies the cyclic doubly-linked list constraint that following the `next`, then `prev` fields always leads back to the initial object. The acyclic constraint `left, right` in the `SleepingProc` role specifies that there are no cycles in the heap involving only `left` and `right` edges. On the other hand, the list of running processes must be cyclic because its nodes can never point to `null`.

The slot constraints specify the complete set of heap aliases for the object. In our example, this implies that no process can be simultaneously running and sleeping.

In general, roles can capture data structure consistency properties such as disjointness and can prevent representation exposure [63, 78]. As a data structure description language, roles can naturally specify trees with additional pointers. Roles can also approximate non-tree data structures like sparse matrices. Because most role constraints are local, it is possible to inductively infer them from data structure instances.

6.1.2 Roles and Procedure Interfaces

Procedures specify the initial and final roles of their parameters. The `suspend` procedure in Figure 6-3, for example, takes two parameters: an object with role `RunningProc p`, and the `SleepingTree s`. The procedure changes the role of the object referenced by `p` to `SleepingProc` whereas the object referenced by `s` retains its original role. To perform the role change, the procedure removes `p` from its `RunningList` data structure and inserts it into the `SleepingTree` data structure `s`. If the procedure fails to perform the insertions or deletions correctly, for instance by leaving an object in both structures, the role analysis will report an error.

6.2 Contributions

This chapter makes the following contributions:

- **Role Concept:** The concept that the state of an object depends on its referencing relationships; specifically, that objects with different heap aliases should be regarded as having different states.

[1]In general, each role definition would specify the static class of objects that can play that role. To simplify the presentation, we assume that all objects are instances of a single class with a set of fields F.

```
role LiveHeader {
  fields next : LiveList | null;
}
role LiveList {
  fields next : LiveList | null,
         proc : RunningProc | SleepingProc;
  slots  LiveList.next | LiveHeader.next;
  acyclic next;
}
role RunningHeader {
  fields next : RunningProc | RunningHeader,
         prev : RunningProc | RunningHeader;
  slots  RunningHeader.next | RunningProc.next,
         RunningHeader.prev | RunningProc.prev;
  identities next.prev, prev.next;
}
role RunningProc {
  fields next : RunningProc | RunningHeader,
         prev : RunningProc | RunningHeader;
  slots  RunningHeader.next | RunningProc.next,
         RunningHeader.prev | RunningProc.prev,
         LiveList.proc;
  identities next.prev, prev.next;
}
role SleepingTree {
  fields root : SleepingProc | null,
  acyclic left, right;
}
role SleepingProc {
  fields left  : SleepingProc | null,
         right : SleepingProc | null;
  slots SleepingProc.left | SleepingProc.right |
        SleepingTree.root;
        LiveList.proc;
  acyclic left, right;
}
role DeadProc {  }
```

Figure 6-2: Role Definitions for a Scheduler

```
procedure suspend(p : RunningProc ->> SleepingProc,
                  s : SleepingTree)
local pp, pn, r;
{
   pp = p.prev;   pn = p.next;
   r = s.root;
   p.prev = null; p.next = null;
   pp.next = pn;  pn.prev = pp;
   s.root = p;    p.left = r;
   setRole(p : SleepingProc);
}
```

Figure 6-3: Suspend Procedure

- **Role Semantics and its Consequences:** It presents a semantics of a language for defining roles. The programmer can use this language to express data structure invariants and properties such as participation of objects in data structures. We show how roles can be used to control the aliasing of objects, and express reachability properties. We show certain decidability and undecidability results for roles.
- **Programming Model:** It presents a set of role consistency rules. These rules give a programming model for changing the role of an object and the circumstances under which roles can be temporarily violated.
- **Procedure Interface Specification Language:** It presents a language for specifying the initial context and effects of each procedure. The effects summarize the actions of the procedure in terms of the references it changes and the regions of the heap that it affects.
- **Role Analysis Algorithm:** It presents an algorithm for verifying that the program respects the constraints given by a set of role definitions and procedure specifications. The algorithm uses a data-flow analysis to infer intermediate referencing relationships between objects, allowing the programmer to focus on role changes and procedure interfaces. The analysis can verify acyclicity constraints even if they are temporarily violated. The interprocedural analysis verifies read effects as well as "may" and "must" write effects by maintaining a fine grained mapping between the current heap and the initial context of the procedure.

6.3 Outline of the Chapter

The rest of the chapter is organized as follows.

In Section 6.4 we introduce the representation of program heap (6.4.1) and the representation of role constraints introduced by the role definitions (6.4.1). We formally define the semantics of roles by giving a criterion for a heap to satisfy the role constraints (6.4.1). We then highlight some application level properties that can be specified using roles (6.4.2) and give examples of using roles to describe data structures. We give a list of properties (6.4.3) that show how roles help control aliasing while giving more flexibility than linear type systems. We show how to deduce reachability properties from role constraints and give a criterion for a set of roles to define a tree. A more detailed study of the constraints expressible using roles is delegated to Appendix 6.9, where we prove decidability of the satisfiability problem for a class of role constraints (6.9.1), and undecidability of the model inclusion for role definitions (6.9.2).

In Section 6.5 we introduce a programming model that enables role definitions to be integrated with the program. We introduce a core programming language with procedures (6.5.1) and give its operational semantics (6.5.2). Next we introduce the notion of onstage and offstage nodes (6.5.3) which defines the criterion for temporary violations of role constraints by generalizing heap consistency from (6.4.1). As part of the programming model we introduce restrictions on programs that simplify later analysis and ensure role consistency across procedure calls (6.5.4). We give the preconditions for transitions of the operational semantics that formalize role consistency. We then introduce an instrumented semantics that gives the programmer complete control over the assignment of roles to objects (6.5.5). This completes the description of the programming model, which is verified by the role analysis.

We present the intraprocedural role analysis in Section 6.6. We define the abstract representation of concrete heaps called role graphs and specify the abstraction relation (6.6.1). We then define transfer functions for the role analysis (6.6.2). This includes the expansion relation (6.6.2) used to instantiate nodes from offstage to onstage using instantiation (6.6.2) and split (6.6.2). We model the movement of nodes offstage using the contraction relation (6.6.2). We also describe the checks that the role analysis performs on role graphs to ensure that the program respects the programming model (6.6.2, 6.6.2).

In Section 6.7 we generalize the role analysis to the interprocedural case. We first introduce procedure interface specification language (6.7.1) that describes initial context (6.7.1) and effects (6.7.1) of each procedure. We give examples of procedure interfaces and define the semantics of initial contexts (6.7.1) and effects (6.7.1). The interprocedural analysis extends the intraprocedural analysis from Section 6.6 by verifying that each procedure respects its specification (6.7.2) and by instantiating procedure specifications to analyze call sites (6.7.3). The verification of transfer relations uses a fine grained mapping between nodes of the role graph at each program point and nodes of the initial context. The analysis of call sites needs to establish the mapping between the current role graphs and callee's initial context (6.7.3), instantiate callee's effects (6.7.3) and then reconstruct the roles of modified non-parameter nodes (6.7.3).

In Section 6.8 we present the extensions of the basic role framework described in previous chapters. These extensions allow a statically unbounded number of heap references to objects (6.8.1), roles defined by references from local variables, non-incremental changes to the role assignment (6.8.4), and roles for specifying partial information about object's fields and aliases (6.8.5). The last section also outlines a subtyping criterion for partial roles.

In Section 6.10 we compare our work to the previous typestate systems, the proposals to control the aliasing in object oriented programming and the term roles as used in object modeling and database community. We compare our role analysis with program verification and analysis techniques for dynamically allocated data structures. Section 6.11 concludes the chapter.

6.4 Roles as a Constraint Specification Language

In this chapter we introduce the formal semantics of roles. We then show how to use roles to specify properties of objects and data structures.

6.4.1 Abstract Syntax and Semantics of Roles

In this section, we precisely define what it means for a given heap to satisfy a set of role definitions. In subsequent sections we will use this definition as a starting point for a programming model and role analysis.

Heap Representation

We represent a concrete program heap as a finite directed graph H_c with $\mathsf{nodes}(H_c)$ representing objects of the heap and labelled edges representing heap references. A graph edge $\langle o_1, f, o_2 \rangle \in H_c$ denotes a reference with field name f from object o_1 to object o_2. To simplify the presentation, we fix a global set of fields F and assume that all objects have the set of fields F.

Role Representation

Let R denote the set of roles used in role definitions, null_R be a special symbol always denoting a null object null_c, and let $R_0 = R \cup \{\mathsf{null}_R\}$. We represent each role as the conjunction of the following four kinds of constraints:

- **Fields:** For every field name $f \in F$ we introduce a function $\mathsf{field}_f : R \to 2^{R_0}$ denoting the set of roles that objects of role $r \in R$ can reference through field f. A field f of role r can be null if and only if $\mathsf{null}_R \in \mathsf{field}_f(r)$. The explicit use of null_R and the possibility to specify a set of alternative roles for every field allows roles to express both may and must referencing relationships.

- **Slots:** Every role r has $\mathsf{slotno}(r)$ slots. A slot $\mathsf{slot}_k(r)$ of role $r \in R$ is a subset of $R \times F$. Let o be an object of role r and o' an object of role r'. A reference $\langle o', f, o \rangle \in H_c$ can fill a slot k of object o if and only if $\langle r', f \rangle \in \mathsf{slot}_k(r)$. An object with role r must have each of its slots filled by exactly one reference.

- **Identities:** Every role $r \in R$ has a set of $\mathsf{identities}(r) \subseteq F \times F$. Identities are pairs of fields $\langle f, g \rangle$ such that following reference f on object o and then returning on reference g leads back to o.
- **Acyclicities:** Every role $r \in R$ has a set $\mathsf{acyclic}(r) \subseteq F$ of fields along which cycles are forbidden.

Role Semantics

We define the semantics of roles as a conjunction of invariants associated with role definitions. A *concrete role assignment* is a map $\rho_c : \mathsf{nodes}(H_c) \to R_0$ such that $\rho_c(\mathsf{null}_c) = \mathsf{null}_R$.

Definition 1 *Given a set of role definitions, we say that heap H_c is* role consistent *iff there exists a role assignment $\rho_c : \mathsf{nodes}(H_c) \to R_0$ such that for every $o \in \mathsf{nodes}(H_c)$ the predicate $\mathsf{locallyConsistent}(o, H_c, \rho_c)$ is satisfied. We call any such role assignment ρ_c a* valid *role assignment.*

The predicate $\mathsf{locallyConsistent}(o, H_c, \rho_c)$ formalizes the constraints associated with role definitions.

Definition 2 $\mathsf{locallyConsistent}(o, H_c, \rho_c)$ *iff all of the following conditions are met. Let $r = \rho_c(o)$.*

1) *For every field $f \in F$ and $\langle o, f, o' \rangle \in H_c$, $\rho_c(o') \in \mathsf{field}_f(r)$.*
2) *Let $\{\langle o_1, f_1 \rangle, \ldots, \langle o_k, f_k \rangle\} = \{\langle o', f \rangle \mid \langle o', f, o \rangle \in H_c\}$ be the set of all aliases of node o. Then $k = \mathsf{slotno}(r)$ and there exists some permutation p of the set $\{1, \ldots, k\}$ such that $\langle \rho_c(o_i), f_i \rangle \in \mathsf{slot}_{p_i}(r)$ for all i.*
3) *If $\langle o, f, o' \rangle \in H_c$, $\langle o', g, o'' \rangle \in H_c$, and $\langle f, g \rangle \in \mathsf{identities}(r)$, then $o = o''$.*
4) *It is not the case that graph H_c contains a cycle $o_1, f_1, \ldots, o_s, f_s, o_1$ where $o_1 = o$ and $f_1, \ldots, f_s \in \mathsf{acyclic}(r)$*

Note that a role consistent heap may have multiple valid role assignments ρ_c. However, in each of these role assignments, every object o is assigned exactly one role $\rho_c(o)$. The existence of a role assignment ρ_c with the property $\rho_c(o_1) \neq \rho_c(o_2)$ thus implies $o_1 \neq o_2$. This is just one of the ways in which roles make aliasing more predictable.

6.4.2 Using Roles

Roles capture important properties of the objects and provide useful information about how the actions of the program affect those properties.

- **Consistency Properties:** Roles can ensure that the program respects application - level data structure consistency properties. The roles in our process scheduler, for example, ensure that a process cannot be simultaneously sleeping and running.

- **Interface Changes:** In many cases, the interface of an object changes as its referencing relationships change. In our process scheduler, for example, only running processes can be suspended. Because procedures declare the roles of their parameters, the role system can ensure that the program uses objects correctly even as the object's interface changes.

- **Multiple Uses:** Code factoring minimizes code duplication by producing general-purpose classes (such as the Java Vector and Hashtable classes) that can be used in a variety of contexts. But this practice obscures the different purposes that different instances of these classes serve in the computation. Because each instance's purpose is usually reflected in its relationships with other objects, roles can often recapture these distinctions.

- **Correlated Relationships:** In many cases, groups of objects cooperate to implement a piece of functionality. Standard type declarations provide some information about these collaborations by identifying the points-to relationships between related objects at the granularity of classes. But roles can capture a much more precise notion of cooperation, because they track correlated state changes of related objects.

Programmers can use roles for specifying the membership of objects in data structures and the structural invariants of data structures. In both cases, the slot constraints are essential.

When used to describe membership of an object in a data structure, slots specify the source of the alias from a data structure node that stores the object. By assigning different sets of roles to data structures used at different program points, it is possible to distinguish nodes stored in different data structure instances. As an object moves between data structures, the role of the object changes appropriately to reflect the new source of the alias.

When describing nodes of data structures, slot constraints specify the aliasing constraints of nodes; this is enough to precisely describe a variety of data structures and approximate many others. Property 16 below shows how to identify trees in role definitions even if tree nodes have additional aliases from other sets of nodes. It is also possible to define nodes which make up a compound data structure linked via disjoint sets of fields, such as threaded trees, sparse matrices and skip lists.

Example 3 The following role definitions specify a sparse matrix of width and height at least 3. These definitions can be easily constructed from a sketch of a sparse matrix in Figure 6-4.

```
role A1 {
  fields x : A2, y : A4;
  acyclic x, y;
}
role A2 {
  fields x : A2 | A3, y : A5;
```

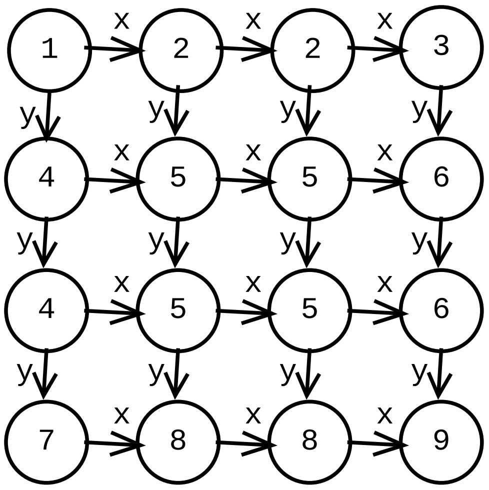


Figure 6-4: Roles of Nodes of a Sparse Matrix

```
  slots A1.x | A2.x;
  acyclic x, y;
}
role A3 {
  fields y : A6;
  slots A2.x;
  acyclic x, y;
}
role A4 {
  fields x : A5, y : A4 | A7;
  slots A1.y | A4.y;
  acyclic x, y;
}
role A5 {
  fields x : A5 | A6, y : A5 | A8;
  slots A4.x | A5.x, A2.y | A5.y;
  acyclic x, y;
}
role A6 {
  fields y : A6 | A9;
  slots A5.x, A3.y | A6.y;
  acyclic x, y;
}
role A7 {
  fields x : A8;
  slots A4.y;
```

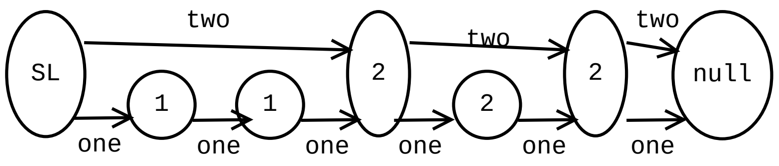


Figure 6-5: Sketch of a Two-Level Skip List

```
  acyclic x, y;
}
role A8 {
  fields x : A8 | A9;
  slots A7.x | A8.x, A5.y;
  acyclic x, y;
}
role A9 {
  slots A8.x, A6.y;
  acyclic x, y;
}
```

△

Example 4 We next give role definitions for a two-level skip list [160] sketched in Figure 6-5.

```
role SkipList {
  fields one : OneNode | TwoNode | null;
         two : TwoNode | null;
}
role OneNode {
  fields one : OneNode | TwoNode | null;
         two : null;
  slots OneNode.one | TwoNode.one | SkipList.one;
  acyclic one, two;
}
role TwoNode {
  fields one : OneNode | TwoNode | null;
         two : TwoNode | null;
  slots OneNode.one | TwoNode.one | SkipList.one,
        TwoNode.two | SkipList.two;
  acyclic one, two;
}
```

△

6.4.3 Some Simple Properties of Roles

In this section we identify some of the invariants expressible using sets of mutually recursive role definitions. Some further properties of roles are given in Appendix 6.9.

The following properties show some of the ways role specifications make object aliasing more predictable. They are an immediate consequence of the semantics of roles.

Property 5 *(Role Disjointness)*
If there exists a valid role assignment ρ_c for H_c such that $\rho(o_1) \neq \rho(o_2)$, then $o_1 \neq o_2$.

The previous property gives a simple criterion for showing that objects o_1 and o_2 are unaliased: find a valid role assignment which assigns different roles to o_1 and o_2. This use of roles generalizes the use of static types for pointer analysis [82]. Since roles create a finer partition of objects than a typical static type system, their potential for proving absence of aliasing is even larger.

Property 6 *(Disjointness Propagation)*
If $\langle o_1, f, o_2\rangle, \langle o_3, g, o_4\rangle \in H_c$, $o_1 \neq o_3$, and there exists a valid role assignment ρ_c for H_c such that $\rho_c(o_2) = \rho_c(o_4) = r$ but $\mathsf{field}_f(r) \cap \mathsf{field}_g(r) = \emptyset$, then $o_2 \neq o_4$.

Property 7 *(Generalized Uniqueness)*
If $\langle o_1, f, o_2\rangle, \langle o_3, g, o_4\rangle \in H_c$, $o_1 \neq o_3$, and there exists a role assignment ρ_c such that $\rho_c(o_2) = \rho_c(o_4) = r$, but there are no indices $i \neq j$ such that $\langle \rho_c(o_1), f\rangle \in \mathsf{slot}_i(r)$ and $\langle \rho_c(o_2), g\rangle \in \mathsf{slot}_j(r)$ then $o_2 \neq o_4$.

A special case of Property 7 occurs when $\mathsf{slotno}(r) = 1$; this constrains all references to objects of role r to be unique.

Role definitions induce a role reference diagram RRD which captures some, but not all, role constraints.

Definition 8 *(Role Reference Diagram)*
Given a set of definitions of roles R, a role reference diagram RRD *is is a directed graph with nodes R_0 and labelled edges defined by*

$$\begin{aligned} \mathsf{RRD} = \; & \{\langle r, f, r'\rangle \mid r' \in \mathsf{field}_f(r) \textit{ and } \exists i \;\; \langle r, f\rangle \in \mathsf{slot}_i(r')\} \\ & \cup \{\langle r, f, \mathsf{null}_R\rangle \mid \mathsf{null}_R \in \mathsf{field}_f(r)\} \end{aligned}$$

Each role reference diagram is a refinement of the corresponding class diagram in a statically typed language, because it partitions classes into multiple roles according to their referencing relationships. The sets $\rho_c^{-1}(r)$ of objects with role r change during program execution, reflecting the changing referencing relationships of objects.

Role definitions give more information than a role reference diagram. Slot constraints specify not only that objects of role r_1 can reference objects of role r_2 along field f, but also give cardinalities on the number of references from other objects. In addition, role definitions include identity and acyclicity constraints, which are not present in role reference diagrams.

Property 9 *Let ρ_c be any valid role assignment. Define*

$$G = \{\langle \rho_c(o_1), f, \rho_c(o_2)\rangle \mid \langle o_1, f, o_2\rangle \in H_c\}$$

Then G is a subgraph of RRD.

It follows from Property 9 that roles give an approximation of may-reachability among heap objects.

Property 10 *(May Reachability)*
If there is a valid role assignment $\rho_c : \mathsf{nodes}(H_c) \to R_0$ such that $\rho_c(o_1) \neq \rho_c(o_2)$ where $o_1, o_2 \in \mathsf{nodes}(H_c)$ and there is no path from $\rho_c(o_1)$ to $\rho_c(o_2)$ in the role reference diagram RRD*, then there is no path from o_1 to o_2 in H_c.*

The next property shows the advantage of explicitly specifying null references in role definitions. While the ability to specify acyclicity is provided by the `acyclic` constraint, it is also possible to indirectly specify must-cyclicity.

Property 11 *(Must Cyclicity)*
Let $F_0 \subseteq F$ and $R_{\mathsf{CYC}} \subseteq R$ be a set of nodes in the role reference diagram RRD *such that for every node $r \in R_{\mathsf{CYC}}$, if $\langle r, f, r'\rangle \in \mathsf{RRD}$ then $r' \in R_{\mathsf{CYC}}$. If ρ_c is a valid role assignment for H_c, then every object $o_1 \in H_c$ with $\rho_c(o_1) \in R_{\mathsf{CYC}}$ is a member of a cycle in H_c with edges from F_0.*

The following property shows that roles can specify a form of must-reachability among the sets of objects with the same role.

Property 12 *(Downstream Path Termination)*
Assume that for some set of fields $F_0 \subseteq F$ there are sets of nodes $R_{\mathsf{INTER}} \subseteq R$, $R_{\mathsf{FINAL}} \subseteq R_0$ of the role reference diagram RRD *such that for every node $r \in R_{\mathsf{INTER}}$:*

1. $F_0 \subseteq \mathsf{acyclic}(r)$
2. *if $\langle r, f, r'\rangle \in \mathsf{RRD}$ for $f \in F_0$, then $r' \in R_{\mathsf{INTER}} \cup R_{\mathsf{FINAL}}$*

Let ρ_c be a valid role assignment for H_c. Then every path in H_c starting from an object o_1 with role $\rho_c(o_1) \in R_{\mathsf{INTER}}$ and containing only edges labelled with F_0 is a prefix of a path that terminates at some object o_2 with $\rho_c(o_2) \in R_{\mathsf{FINAL}}$.

Property 13 *(Upstream Path Termination)*
Assume that for some set of fields $F_0 \subseteq F$ there are sets of nodes $R_{\mathsf{INTER}} \subseteq R$, $R_{\mathsf{INIT}} \subseteq R_0$ of the role reference diagram RRD *such that for every node $r \in R_{\mathsf{INTER}}$:*

1. $F_0 \subseteq \mathsf{acyclic}(r)$
2. *if $\langle r', f, r\rangle \in \mathsf{RRD}$ for $f \in F_0$, then $r' \in R_{\mathsf{INTER}} \cup R_{\mathsf{INIT}}$*

Let ρ_c be a valid role assignment for H_c. Then every path in H_c terminating at an object o_2 with $\rho_c(o_2) \in R_{\mathsf{INTER}}$ and containing only edges labelled with F_0 is a suffix of a path which started at some object o_1, where $\rho_c(o_1) \in R_{\mathsf{INIT}}$.

We next describe the conditions that guarantee the existence at least one path in the heap, rather than stating the properties of all paths as in Properties 12 and 13.

Property 14 *(Downstream Must Reachability)*
Assume that for some set of fields $F_0 \subseteq F$ *there are sets of roles* $R_{\mathsf{INTER}} \subseteq R$, $R_{\mathsf{FINAL}} \subset R_0$ *of the role reference diagram* RRD *such that for every node* $r \in R_{\mathsf{INTER}}$*:*

1. $F_0 \subseteq \mathsf{acyclic}(r)$
2. *there exists* $f \in F_0$ *such that* $\mathsf{field}_f(r) \subseteq R_{\mathsf{INTER}} \cup R_{\mathsf{FINAL}}$

Let ρ_c *be a valid role assignment for* H_c*. Then for every object* o_1 *with* $\rho_c(o_1) \in R_{\mathsf{INTER}}$ *there is a path in* H_c *with edges from* F_0 *from* o_1 *to some object* o_2 *where* $\rho_c(o_2) \in R_{\mathsf{FINAL}}$*.*

Property 15 *(Upstream Must Reachability)*
Assume that for some set of fields $F_0 \subseteq F$ *there are sets of nodes* $R_{\mathsf{INTER}} \subseteq R$, $R_{\mathsf{INIT}} \subseteq R$ *of the role reference diagram* RRD *such that for every node* $r \in R_{\mathsf{INTER}}$*:*

1. $F_0 \subseteq \mathsf{acyclic}(r)$
2. *there exists* k *such that* $\mathsf{slot}_k(r) \subseteq (R_{\mathsf{INTER}} \cup R_{\mathsf{INIT}}) \times F$

Let ρ_c *be a valid role assignment for* H_c*. Then for every object* o_2 *with* $\rho_c(o_2) \in R_{\mathsf{INTER}}$ *there is a path in* H_c *from some object* o_1 *with* $\rho_c(o_1) \in R_{\mathsf{INIT}}$ *to the object* o_2*.*

Trees are a class of data structures especially suited for static analysis. Roles can express graphs that are not trees, but it is useful to identify trees as certain sets of mutually recursive role definitions.

Property 16 *(Treeness)*
Let $R_{\mathsf{TREE}} \subseteq R$ *be a set of roles and* $F_0 \subseteq F$ *set of fields such that for every* $r \in R_{\mathsf{TREE}}$

1. $F_0 \subseteq \mathsf{acyclic}(r)$
2. $|\{i \mid \mathsf{slot}_i(r) \cap (R_{\mathsf{TREE}} \times F_0) \neq \emptyset\}| \leq 1$

Let ρ_c *be a valid role assignment for* H_c *and*

$$S \subseteq \{\langle n_1, f, n_2\rangle \mid \langle n_1, f, n_2\rangle \in H_c, \rho(n_1), \rho(n_2) \in R_{\mathsf{TREE}}, f \in F_0\}$$

Then S *is a set of trees.*

6.5 A Programming Model

In this section we define what it means for an execution of a program to respect the role constraints. This definition is complicated by the need to allow the program to temporarily violate the role constraints during data structure manipulations. Our approach is to let the program violate the constraints for objects referenced by local variables or parameters, but require all other objects to satisfy the constraints.

We first present a simple imperative language with dynamic object allocation and give its operational semantics. We then specify additional statement preconditions that enforce the role consistency requirements.

$$\begin{array}{rcl} \mathtt{if}\ \mathsf{t}\ \mathsf{stat}_1\ \mathsf{stat}_2 & \equiv & (\mathtt{test(t)};\mathsf{stat}_1)\,|\,(\mathtt{test(!t)};\mathsf{stat}_2) \\ \mathtt{while}\ \mathsf{t}\ \mathsf{stat} & \equiv & (\mathtt{test(t)};\mathsf{stat})*;\mathtt{test(!t)} \end{array}$$

Figure 6-6: Syntactic Sugar for `if` and `while`

6.5.1 A Simple Imperative Language

Our core language contains, as basic statements, Load (`x=y.f`), Store (`x.f=y`), Copy (`x=y`), and New (`x=new`). All variables are references to objects in the global heap and all assignments are reference assignments. We use an elementary `test` statement combined with nondeterministic choice and iteration to express `if` and `while` statement, using the usual translation [117, 28] given in Figure 6-6. We represent the control flow of programs using control-flow graphs.

A program is a collection of procedures $\mathsf{proc} \in \mathsf{Proc}$. Procedures change the global heap but do not return values. Every procedure proc has a list of parameters $\mathsf{param}(\mathsf{proc}) = \{\mathsf{param}_i(\mathsf{proc})\}_i$ and a list of local variables $\mathsf{local}(\mathsf{proc})$. We use $\mathsf{var}(\mathsf{proc})$ to denote $\mathsf{param}(\mathsf{proc}) \cup \mathsf{local}(\mathsf{proc})$. A procedure definition specifies the initial role $\mathsf{preR}_k(\mathsf{proc})$ and the final role $\mathsf{postR}_k(\mathsf{proc})$ for every parameter $\mathsf{param}_k(\mathsf{proc})$. We use proc_j for indices $j \in \mathcal{N}$ to denote activation records of procedure proc. We further assume that there are no modifications of parameter variables so every parameter references the same object throughout the lifetime of procedure activation.

Example 17 The following `kill` procedure removes a process from both the doubly linked list of running processes and the list of all active processes. This is indicated by the transition from `RunningProc` to `DeadProc`.

```
procedure kill(p : RunningProc ->> DeadProc,
               l : LiveHeader)
local prev, current, cp, nxt, lp, ln;
{
  // find 'p' in 'l'
  prev = l; current = l.next;
  cp = current.proc;
  while (cp != p) {
    prev = current;
    current = current.next;
    cp = current.proc;
  }
  // remove 'current' and 'p' from active list
  nxt = current.next;
  prev.next = nxt; current.
  current.proc = null;
  setRole(current : IsolatedCell);
  // remove 'p' from running list
  lp = p.prev;   ln = p.next;
```

Statement	Transition	Constraints	Role Consistency
p : `x=y.f`	$\langle p@\mathsf{proc}_i; s, H_c \uplus \{\langle \mathsf{proc}_i, \mathtt{x}, o_x\rangle\}\rangle \rightarrow$ $\langle p'@\mathsf{proc}_i; s, H'_c\rangle$	$\mathtt{x}, \mathtt{y} \in \mathsf{local}(\mathsf{proc})$, $\langle \mathsf{proc}_i, \mathtt{y}, o_y\rangle, \langle o_y, \mathtt{f}, o_f\rangle \in H_c$, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$, $H'_c = H_c \uplus \{\mathsf{proc}_i, \mathtt{x}, o_f\}$	$\mathsf{accessible}(o_f, \mathsf{proc}_i, H_c)$, $\mathsf{con}(H'_c, \mathsf{offstage}(H'_c))$
p : `x.f=y`	$\langle p@\mathsf{proc}_i; s, H_c \uplus \{\langle o_x, f, o_f\rangle\}\rangle \rightarrow$ $\langle p'@\mathsf{proc}_i; s, H'_c\rangle$	$\mathtt{x}, \mathtt{y} \in \mathsf{local}(\mathsf{proc})$, $\langle \mathsf{proc}_i, \mathtt{x}, o_x\rangle, \langle \mathsf{proc}_i, \mathtt{y}, o_y\rangle \in H_c$, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$, $H'_c = H_c \uplus \{\langle o_x, f, o_y\rangle\}$	$o_f \in \mathsf{onstage}(H_c, \mathsf{proc}_i)$ $\mathsf{con}(H'_c, \mathsf{offstage}(H'_c))$
p : `x=y`	$\langle p@\mathsf{proc}_i; s, H_c \uplus \{\langle \mathsf{proc}_i, \mathtt{x}, o_x\rangle\}\rangle \rightarrow$ $\langle p'@\mathsf{proc}_i; s, H'_c\rangle$	$\mathtt{x} \in \mathsf{local}(\mathsf{proc})$, $\mathtt{y} \in \mathsf{var}(\mathsf{proc})$, $\langle \mathsf{proc}_i, \mathtt{y}, o_y\rangle \in H_c$, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$, $H'_c = H_c \uplus \{\langle \mathsf{proc}_i, \mathtt{x}, o_y\rangle\}$	$\mathsf{con}(H'_c, \mathsf{offstage}(H'_c))$
p : `x=new`	$\langle p@\mathsf{proc}_i; s, H_c \uplus \{\langle \mathsf{proc}_i, \mathtt{x}, o_x\rangle\}\rangle \rightarrow$ $\langle p'@\mathsf{proc}_i; s, H'_c\rangle$	$\mathtt{x} \in \mathsf{local}(\mathsf{proc})$, o_n fresh, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$, $H'_c = H_c \uplus \{\langle \mathsf{proc}_i, \mathtt{x}, o_n\rangle\} \uplus \mathsf{nulls}$, $\mathsf{nulls} = \{o_n\} \times F \times \{\mathsf{null}\}$	$\mathsf{con}(H'_c, \mathsf{offstage}(H'_c))$
p : `test(c)`	$\langle p@\mathsf{proc}_i; s, H_c\rangle \rightarrow$ $\langle p'@\mathsf{proc}_i; s, H_c\rangle$	$\mathsf{satisfied}_c(\mathsf{c}, \mathsf{proc}_i, H_c)$, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$	$\mathsf{con}(H_c, \mathsf{offstage}(H_c))$

$$\mathsf{satisfied}_c(\mathtt{x==y}, \mathsf{proc}_i, H_c) \text{ iff } \{o \mid \langle \mathsf{proc}_i, \mathtt{x}, o\rangle \in H_c\} = \{o \mid \langle \mathsf{proc}_i, \mathtt{y}, o\rangle \in H_c\}$$

$$\mathsf{satisfied}_c(\mathtt{!(x==y)}, \mathsf{proc}_i, H_c) \text{ iff not } \mathsf{satisfied}_c(\mathtt{x==y}, \mathsf{proc}_i, H_c)$$

$$\begin{array}{lcl} \mathsf{accessible}(o, \mathsf{proc}_i, H_c) & := & (\exists p \in \mathsf{param}(\mathsf{proc}) : \langle \mathsf{proc}_i, p, o\rangle \in H_c) \\ & & \text{ or } \ \text{not } (\exists \mathsf{proc}'_j \ \exists v \in \mathsf{var}(\mathsf{proc}') : \langle \mathsf{proc}'_j, v, o\rangle \in H_c) \end{array}$$

Figure 6-7: Semantics of Basic Statements

```
  p.prev = null; p.next = null;
  lp.next = ln;   ln.prev = lp;
  setRole(p : DeadProc);
}
```

△

6.5.2 Operational Semantics

In this section we give the operational semantics for our language. We focus on the first three columns in Figures 6-7 and 6-8; the safety conditions in the fourth column are detailed in Section 6.5.4.

Figure 6-7 gives the small-step operational semantics for the basic statements. We use $A \uplus B$ to denote the union $A \cup B$ where the sets A and B are disjoint. The program state consists of the stack s and the concrete heap H_c. The stack s is a sequence of pairs $p@\mathsf{proc}_i \in \times(\mathsf{Proc} \times \mathcal{N})$, where $p \in N_{\mathsf{CFG}}(\mathsf{proc})$ is a program point, and $\mathsf{proc}_i \in \mathsf{Proc} \times \mathcal{N}$ is an activation record of procedure proc. Program points $p \in N_{\mathsf{CFG}}(\mathsf{proc})$ are nodes of the control-flow graphs. There is one control-flow graph for every procedure proc. An edge of the control-flow graph $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$ indicates that control may transfer from point p to point p'. We write $p : \mathsf{stat}$ to state that program point p contains a statement stat. The control flow graph of each

Statement	Transition	Constraints	Role Consistency
$\texttt{entry} : \_$	$\langle p@\mathsf{proc}_i; s, H_c\rangle \rightarrow$ $\langle p'@\mathsf{proc}_i; s, H_c \uplus \mathsf{nulls}\rangle$	$\mathsf{nulls} = \{\langle \mathsf{proc}_i, v, \mathsf{null}_c\rangle \mid v \in \mathsf{local}(\mathsf{proc}),$ $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$	$\mathsf{con}(H_c, \mathsf{offstage}(H_c))$
$p : \mathsf{proc}'(x_k)_k$	$\langle p@\mathsf{proc}_i; s, H_c\rangle \rightarrow$ $\langle \texttt{entry}@\mathsf{proc}'_j; p'@\mathsf{proc}_i; s, H'_c\rangle$	j fresh in $p@\mathsf{proc}i; s,$ $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc}),$ $o_k : \langle \mathsf{proc}_i, x_k, o_k\rangle \in H_c,$ $H'_c = H_c \uplus \{\langle \mathsf{proc}'_j, p_k, o_k\rangle\}_k,$ $\forall k \;\; p_k = \mathsf{param}_k(\mathsf{proc}')$	$\mathsf{conW}(\mathsf{ra}, H_c, S),$ $\mathsf{ra} = \{\langle o_k, \mathsf{preR}_k(\mathsf{proc}')\rangle\}_k,$ $S = \mathsf{offstage}(H_c) \cup \{o_k\}_k$
$\texttt{exit} : \_$	$\langle p@\mathsf{proc}_i; s, H_c\rangle \rightarrow$ $\langle s, H_c \setminus \mathsf{AF}\rangle$	$\mathsf{AF} = \{\langle \mathsf{proc}_i, v, n\rangle \mid$ $\langle \mathsf{proc}_i, v, n\rangle \in H_c\}$	$\mathsf{conW}(\mathsf{ra}, H_c, S),$ $\mathsf{ra} = \{\langle \mathsf{parnd}_k(\mathsf{proc}_i), \mathsf{postR}_k(\mathsf{proc})\rangle\}_k,$ $S = \mathsf{offstage}(H_c) \cup$ $\{o \mid \langle \mathsf{proc}_i, v, o\rangle \in H_c\}$

$$\mathsf{parnd}_k(\mathsf{proc}_i) = o \text{ where } \langle \mathsf{proc}_i, \mathsf{param}_k(\mathsf{proc}), o\rangle \in H_c$$

Figure 6-8: Semantics of Procedure Call

procedure contains special program points `entry` and `exit` indicating procedure entry and exit, with no statements associated with them. We assume that each condition of a `test` statement is of the form `x==y` or `!(x==y)` where `x` and `y` are either variables or a special constant `null` which always points to the null_c object.

The concrete heap is either an error heap error_c or a non-error heap. A non-error heap $H_c \subseteq N \times F \times N \cup ((\mathsf{Proc} \times \mathcal{N}) \times V \times N)$ is a directed graph with labelled edges, where nodes represent objects and procedure activation records, whereas edges represent heap references and local variables. An edge $\langle o_1, f, o_2\rangle \in N \times F \times N$ denotes a reference from object o_1 to object o_2 via field $f \in F$. An edge $\langle \mathsf{proc}_i, \mathtt{x}, o\rangle \in H_c$ means that local variable `x` in activation record proc_i points to object o.

A load statement `x=y.f` makes the variable `x` point to node o_f, which is referenced by the `f` field of object o_y, which is in turn referenced by variable `y`. A store statement `x.f=y` replaces the reference along field `f` in object o_x by a reference to object o_y that is referenced by `y`. The copy statement `x=y` copies a reference to object o_y into variable `x`. The statement `x=new` creates a new object o_n with all fields initially referencing null_c, and makes `x` point to o_n. The statement `test(c)` allows execution to proceed only if condition `c` is satisfied.

Figure 6-8 shows the semantics of procedure calls. Procedure call pushes new activation record onto stack, inserts it into the heap, and initializes the parameters. Procedure entry initializes local variables. Procedure exit removes the activation record from the heap and the stack.

6.5.3 Onstage and Offstage Objects

At every program point the set $\mathsf{nodes}(H_c)$ of all objects of heap H_c can be partitioned into:

1. **onstage objects** ($\mathsf{onstage}(H_c)$) referenced by a local variable or parameter of some activation frame

$$\begin{aligned}\mathsf{onstage}(H_c, \mathsf{proc}_i) :=& \{o \mid \exists x \in \mathsf{var}(\mathsf{proc}) \\ & \langle \mathsf{proc}_i, x, o\rangle \in H_c\} \\ \mathsf{onstage}(H_c) :=& \bigcup_{\mathsf{proc}_i} \mathsf{onstage}(H_c, \mathsf{proc}_i)\end{aligned}$$

2. **offstage objects** ($\mathsf{offstage}(H_c)$) unreferenced by local or parameter variables

$$\mathsf{offstage}(H_c) := \mathsf{nodes}(H_c) \setminus \mathsf{onstage}(H_c)$$

Onstage objects need not have correct roles. Offstage objects must have correct roles assuming some role assignment for onstage objects.

Definition 18 *Given a set of role definitions and a set of objects $S_c \subseteq \mathsf{nodes}(S_c)$, we say that heap H_c is* role consistent *for S_c, and we write $\mathsf{con}(H_c, S_c)$, iff there exists a role assignment $\rho_c : \mathsf{nodes}(H_c) \to R_0$ such that the $\mathsf{locallyConsistent}(o, H_c, \rho_c, S_c)$ predicate is satisfied for every object $o \in S_c$.*

We define $\mathsf{locallyConsistent}(o, H_c, \rho_c, S_c)$ to generalize the $\mathsf{locallyConsistent}(o, H_c, \rho_c)$ predicate, weakening the acyclicity condition.

Definition 19 $\mathsf{locallyConsistent}(o, H_c, \rho_c, S_c)$ *holds iff conditions 1), 2), and 3) of Definition 2 are satisfied and the following condition holds:*

4') It is not the case that graph H_c contains a cycle $o_1, f_1, \ldots, o_s, f_s, o_1$ such that $o_1 = o$, $f_1, \ldots, f_s \in \mathsf{acyclic}(r)$, and additionally $o_1, \ldots, o_s \in S_c$.

Here S_c is the set of onstage objects that are not allowed to create a cycle whereas objects in $\mathsf{nodes}(H_c) \setminus S_c$ are exempt from the acyclicity condition. The predicates $\mathsf{locallyConsistent}(o, H_c, \rho_c, S_c)$ and $\mathsf{con}(H_c, S_c)$ are monotonic in S_c, so a larger S_c implies a stronger invariant. For $S_c = \mathsf{nodes}(H_c)$, consistency for S_c is equivalent with heap consistency from Definition 1. Note that the role assignment ρ_c specifies roles even for objects $o \in \mathsf{nodes}(H_c) \setminus S_c$. This is because the role of o may influence the role consistency of objects in S_c which are adjacent to o.

At procedure calls, the role declarations for parameters restrict the set of potential role assignments. We therefore generalize $\mathsf{con}(H_c, S_c)$ to $\mathsf{conW}(\mathsf{ra}, H_c, S_c)$, which restricts the set of role assignments ρ_c considered for heap consistency.

Definition 20 *Given a set of role definitions, a heap H_c, a set $S_c \subseteq \mathsf{nodes}(H_c)$, and a partial role assignment $\mathsf{ra} \subseteq S_c \to R$, we say that the heap H_c is* consistent with ra for S_c*, and write $\mathsf{conW}(\mathsf{ra}, H_c, S_c)$, iff there exists a (total) role assignment $\rho_c : \mathsf{nodes}(H_c) \to R_0$ such that $\mathsf{ra} \subseteq \rho_c$ and for every object $o \in S_c$ the predicate $\mathsf{locallyConsistent}(o, H_c, \rho_c, S_c)$ is satisfied.*

6.5.4 Role Consistency

We are now able to precisely state the role consistency requirements that must be satisfied for program execution. The role consistency requirements are in the fourth row of Figures 6-7 and 6-8. We assume the operational semantics is extended with transitions leading to a program state with heap error_c whenever role consistency is violated.

Offstage Consistency

At every program point, we require $\mathsf{con}(H_c, \mathsf{offstage}(H_c))$ to be satisfied. This means that offstage objects have correct roles, but onstage objects may have their role temporarily violated.

Reference Removal Consistency

The Store statement `x.f=y` has the following safety precondition. When a reference $\langle o_x, f, o_f\rangle \in H_c$ for $\langle \mathsf{proc}_j, \mathtt{x}, o_x\rangle \in H_c$, and $\langle o_x, \mathtt{f}, o_f\rangle \in H_c$ is removed from the heap, both o_x and o_f must be referenced from the current procedure activation record. It is sufficient to verify this condition for o_f, as o_x is already onstage by definition. The reference removal consistency condition enables the completion of the role change for o_f after the reference $\langle o_x, f, o_f\rangle$ is removed and ensures that heap references are introduced and removed only between onstage objects.

Procedure Call Consistency

Our programming model ensures role consistency across procedure calls using the following protocol.

A procedure call $\mathsf{proc}'(x_1, ..., x_p)$ in Figure 6-8 requires the role consistency precondition $\mathsf{conW}(\mathsf{ra}, H_c, S_c)$, where the partial role assignment ra requires objects o_k, corresponding to parameters x_k, to have roles $\mathsf{preR}_k(\mathsf{proc}')$ expected by the callee, and $S_c = \mathsf{offstage}(H_c) \cup \{o_k\}_k$ for $\langle \mathsf{proc}_j, x_k, o_k\rangle \in H_c$.

To ensure that the callee proc'_j never observes incorrect roles, we impose an *accessibility condition* for the callee's Load statements (see the fourth column of Figure 6-7). The accessibility condition prohibits access to any object o referenced by some local variable of a stack frame other than proc'_j, unless o is referenced by some parameter of proc'_j. Provided that this condition is not violated, the callee proc'_j only accesses objects with correct roles, even though objects that it does not access may have incorrect roles. In Section 6.7 we show how the role analysis statically ensures that the accessibility condition is never violated.

At the procedure exit point (Figure 6-8), we require correct roles for all objects referenced by the current activation frame proc'_j. This implies that heap operations performed by proc'_j preserve heap consistency for all objects accessed by proc'_j.

Statement	Transition	Constraints	Role Consistency
$p : \texttt{roleCheck}(x_1, \ldots, x_n, \mathsf{ra})$	$\langle p@\mathsf{proc}_i; s, H_c\rangle \rightarrow$ $\langle p'@\mathsf{proc}_i; s, H_c\rangle$	$\langle p, p'\rangle \in E_{\mathsf{CFG}}$	$\mathsf{conW}(\mathsf{ra}, H_c, S)$, $S = \mathsf{offstage}(H_c) \cup \{o \mid \langle \mathsf{proc}_i, x_k, o\rangle \in H_c\}$

Figure 6-9: Operational Semantics of Explicit Role Check

Statement	Transition	Constraints	Role Consistency
$p : \texttt{x=new}$	$\langle p@\mathsf{proc}_i; s, H_c \uplus \{\langle \mathsf{proc}_i, \mathtt{x}, o_x\rangle\}, \rho_c\rangle \rightarrow$ $\langle p'@\mathsf{proc}_i; s, H_c', \rho_c'\rangle$	$\mathtt{x} \in \mathsf{local}(\mathsf{proc})$, o_n fresh, $\langle p, p'\rangle \in E_{\mathsf{CFG}}(\mathsf{proc})$, $H_c' = H_c$ $\uplus\{\langle \mathsf{proc}_i, \mathtt{x}, o_n\rangle\}$ $\uplus\{o_n\} \times F \times \{\mathsf{null}\}$, $\rho_c' = \rho_c[o_n \mapsto \mathsf{unknown}]$	$\mathsf{conW}(\rho_c', H_c', \mathsf{offstage}(H_c'))$
$p :$ $\texttt{setRole(x:r)}$	$\langle p@\mathsf{proc}_i; s, H_c, \rho_c\rangle \rightarrow$ $\langle p'@\mathsf{proc}_i; s, H_c, \rho_c'\rangle$	$\mathtt{x} \in \mathsf{local}(\mathsf{proc}_i)$, $\langle \mathsf{proc}_i, \mathtt{x}, o_x\rangle \in H_c$, $\rho_c' = \rho_c[o_x \mapsto \mathtt{r}]$, $\langle p, p'\rangle \in E_{\mathsf{CFG}}$	$\mathsf{conW}(\rho_c', H_c, \mathsf{offstage}(H_c))$
$p : \mathsf{stat}$	$\langle s, H_c, \rho_c\rangle \rightarrow$ $\langle s', H_c', \rho_c\rangle$	$\langle s, H_c\rangle \rightarrow \langle s', H_c'\rangle$	$P \wedge \mathsf{conW}(\rho_c \cup \mathsf{ra}, H_c'', S)$ for every original condition $P \wedge \mathsf{conW}(\mathsf{ra}, H_c'', S)$

Figure 6-10: Instrumented Semantics

Explicit Role Check

The programmer can specify a stronger invariant at any program point using statement $\texttt{roleCheck}(x_1, \ldots, x_p, \mathsf{ra})$. As Figure 6-9 indicates, `roleCheck` requires the $\mathsf{conW}(\mathsf{ra}, H_c, S_c)$ predicate to be satisfied for the supplied partial role assignment ra where $S_c = \mathsf{offstage}(H_c) \cup \{o_k\}_k$ for objects o_k referenced by given local variables x_k.

6.5.5 Instrumented Semantics

We expect the programmer to have a specific role assignment in mind when writing the program, with this role assignment changing as the statements of the program change the referencing relationships. So when the programmer wishes to change the role of an object, he or she writes a program that brings the object onstage, changes its referencing relationships so that it plays a new role, then puts it offstage in its new role. The roles of other objects do not change.[2]

To support these programmer expectations, we introduce an augmented programming model in which the role assignment ρ_c is conceptually part of the program's state. The role assignment changes only if the programmer changes it explicitly using the `setRole` statement. The augmented programming model has an underlying *instrumented semantics* as opposed to the *original semantics*.

[2]An extension to the programming model supports *cascading role changes* in which a single role change propagates through the heap changing the roles of offstage objects, see Section 6.8.4.

Example 21 The original semantics allows asserting different roles at different program points even if the structure of the heap was not changed, as in the following procedure `foo`.

```
role A1 { fields f : B1; }
role B1 { slots A1.f; }
role A2 { fields f : B2; }
role B2 { slots A2.f; }
procedure foo()
var x, y;
{
  x = new;  y = new;
  x.f = y;
  roleCheck(x,y, x:A1,y:B1);
  roleCheck(x,y, x:A2,y:B2);
}
```

Both role checks would succeed since each of the specified partial role assignments can be extended to a valid role assignment. On the other hand, the role check statement `roleCheck(x,y, x:A1,y:B2)` would fail.

The procedure `foo` in the instrumented semantics can be written as follows.

```
procedure foo()
var x, y;
{
  x = new;  y = new;
  x.f = y;
  setRole(x:A1);  setRole(y:B1);
  roleCheck(x,y, x:A1,y:B1);
  setRole(x:A2);  setRole(y:B2);
  roleCheck(x,y, x:A2,y:B2);
}
```

The `setRole` statement makes the role change of object explicit. $\triangle$

The instrumented semantics extends the concrete heap H_c with a role assignment ρ_c. Figure 6-10 outlines the changes in instrumented semantics with respect to the original semantics. We introduce a new statement `setRole(x:r)`, which modifies a role assignment ρ_c, giving $\rho_c[o_x \mapsto r]$, where o_x is the object referenced by `x`. All statements other than `setRole` preserve the current role assignment. For every consistency condition $\mathsf{conW}(\mathsf{ra}, H_c, S_c)$ in the original semantics, the instrumented semantics uses the corresponding condition $\mathsf{conW}(\rho_c \cup \mathsf{ra}, H_c, S_c)$ and fails if ρ_c is not an extension of ra. Here we consider $\mathsf{con}(H_c, S)$ to be a shorthand for $\mathsf{conW}(\emptyset, H_c, S)$. For example, the new role consistency condition for the Copy statement `x=y` is $\mathsf{conW}(\rho_c, H_c, \mathsf{offstage}(H_c))$. The New statement assigns an identifier $\mathsf{unknown}$ to the newly created object o_n. By definition, a node with $\mathsf{unknown}$ does not satisfy the $\mathsf{locallyConsistent}$ predicate. This means that `setRole` must be used to set a a valid role of o_n before o_n moves offstage.

By introducing an instrumented semantics we are not suggesting an implementation that explicitly stores roles of objects at run-time. We instead use the instrumented semantics as the basis of our role analysis and ensure that all role checks can be statically removed. Because the instrumented semantics is more restrictive than the original semantics, our role analysis is a conservative approximation of both the instrumented semantics and the original semantics.

6.6 Intraprocedural Role Analysis

This section presents an intraprocedural role analysis algorithm. The goal of the role analysis is to statically verify the role consistency requirements described in the previous section.

The key observation behind our analysis algorithm is that we can incrementally verify role consistency of the entire concrete heap H_c by ensuring role consistency for every node when it goes offstage. This allows us to represent the statically unbounded offstage portion of the heap using summary nodes with "may" references. In contrast, we use a "must" interpretation for references from and to onstage nodes. The exact representation of onstage nodes allows the analysis to verify role consistency in the presence of temporary violations of role constraints.

Our analysis representation is a graph in which nodes represent objects and edges represent references between objects. There are two kinds of nodes: *onstage nodes* represent onstage objects, with each onstage node representing one onstage object; and *offstage nodes*, with each offstage node corresponding to a set of objects that play that role. To increase the precision of the analysis, the algorithm occasionally generates multiple offstage nodes that represent disjoint sets of objects playing the same role. Distinct offstage objects with the same role r represent disjoint sets of objects of role r with different reachability properties from onstage nodes.

We frame role analysis as a data-flow analysis operating on a distributive lattice $\mathcal{P}(\mathsf{RoleGraphs})$ of sets of role graphs with set union $\cup$ as the join operator. This section focuses on the intraprocedural analysis. We use proc_c to denote the topmost activation record in a concrete heap H_c. In Section 6.7 we generalize the algorithm to the compositional interprocedural analysis.

6.6.1 Abstraction Relation

Every data-flow fact $\mathcal{G} \subseteq \mathsf{RoleGraphs}$ is a set of role graphs $G \in \mathcal{G}$. Every role graph $G \in \mathsf{RoleGraphs}$ is either a bottom role graph $\bot_G$ representing the set of all concrete heaps (including error_c), or a tuple $G = \langle H, \rho, K \rangle$ representing non-error concrete heaps, where

- $H \subseteq N \times F \times N$ is the abstract heap with nodes N representing objects and fields F. The abstract heap H represents heap references $\langle n_1, f, n_2 \rangle$ and variables of the currently analyzed procedure $\langle \mathsf{proc}, x, n \rangle$ where $x \in \mathsf{local}(\mathsf{proc})$. Null references are represented as references to abstract node null. We define abstract onstage nodes $\mathsf{onstage}(H) = \{n \mid \langle \mathsf{proc}, x, n\rangle \in H, x \in \mathsf{local}(\mathsf{proc}) \cup \mathsf{param}(\mathsf{proc})\}$ and abstract offstage nodes $\mathsf{offstage}(H) = \mathsf{nodes}(H) \setminus \mathsf{onstage}(H) \setminus \{\mathsf{proc}, \mathsf{null}\}$.

- $\rho : \mathsf{nodes}(H) \to R_0$ is an abstract role assignment, $\rho(\mathsf{null}) = \mathsf{null}_R$;
- $K : \mathsf{nodes}(H) \to \{i, s\}$ indicates the kind of each node; when $K(n) = i$, then n is an individual node representing at most one object, and when $K(n) = s$, n is a summary node representing zero or more objects. We require $K(\mathsf{proc}) = K(\mathsf{null}) = i$, and require all onstage nodes to be individual, $K[\mathsf{onstage}(H)] = \{i\}$.

The abstraction relation α relates a pair $\langle H_c, \rho_c\rangle$ of concrete heap and concrete role assignment with an abstract role graph G.

Definition 22 *We say that an abstract role graph G represents concrete heap H_c with role assignment ρ_c, and write $\langle H_c, \rho_c\rangle \,\alpha\, G$, iff $G = \bot_G$ or: $H_c \neq \mathsf{error}_c$, $G = \langle H, \rho, K\rangle$, and there exists a function $h : \mathsf{nodes}(H_c) \to \mathsf{nodes}(H)$ such that*

1) *H_c is role consistent: $\mathsf{conW}(\rho_c, H_c, \mathsf{offstage}(H_c))$,*
2) *identity relations of onstage nodes with offstage nodes hold: if $\langle o_1, f, o_2\rangle \in H_c$ and $\langle o_2, g, o_3\rangle \in H_c$ for $o_1 \in \mathsf{onstage}(H_c)$, $o_2 \in \mathsf{offstage}(H_c)$, and $\langle f, g\rangle \in \mathsf{identities}(\rho_c(o_1))$, then $o_3 = o_1$;*
3) *h is a graph homomorphism: if $\langle o_1, f, o_2\rangle \in H_c$ then $\langle h(o_1), f, h(o_2)\rangle \in H$;*
4) *an individual node represents at most one concrete object: $K(n) = i$ implies $|h^{-1}(n)| \leq 1$;*
5) *h is bijection on edges which originate or terminate at onstage nodes: if $\langle n_1, f, n_2\rangle \in H$ and $n_1 \in \mathsf{onstage}(H)$ or $n_2 \in \mathsf{onstage}(H)$, then there exists exactly one $\langle o_1, f, o_2\rangle \in H_c$ such that $h(o_1) = n_1$ and $h(o_2) = n_2$;*
6) *$h(\mathsf{null}_c) = \mathsf{null}$ and $h(\mathsf{proc}_c) = \mathsf{proc}$;*
7) *the abstract role assignment ρ corresponds to the concrete role assignment: $\rho_c(o) = \rho(h(o))$ for every object $o \in \mathsf{nodes}(H_c)$.*

Note that the error heap error_c can be represented only by the bottom role graph $\bot_G$. The analysis uses $\bot_G$ to indicate a potential role error.

Condition 3) implies that role graph edges are a conservative approximation of concrete heap references. These edges are in general "may" edges. Hence it is possible for an offstage node n that $\langle n, f, n_1\rangle$, $\langle n, f, n_2\rangle \in H$ for $n_1 \neq n_2$. This cannot happen when $n \in \mathsf{onstage}(H)$ because of 5). Another consequence of 5) is that an edge in H from an onstage node n_0 to a summary node n_s implies that n_s represents at least one object. Condition 2) strengthens 1) by requiring certain identity constraints for onstage nodes to hold, as explained in Section 6.6.2.

Example 23 Consider the following role declaration for an acyclic list.

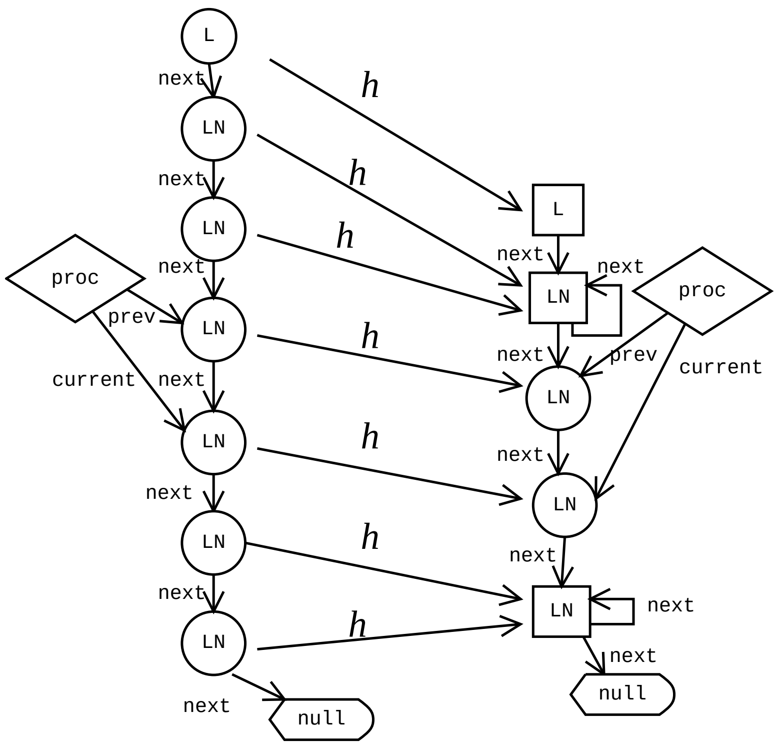


Figure 6-11: Abstraction Relation

```
role L { // List header
  fields first : LN | null;
}
role LN { // List node
  fields next : LN | null;
  slots LN.next | L.first;
  acyclic next;
}
```

Figure 6-11 shows a role graph and one of the concrete heaps represented by the role graph via homomorphism h. There are two local variables, `prev` and `current`, referencing distinct onstage objects. Onstage objects are isomorphic to onstage nodes in the role graph. In contrast, there are two objects mapped to each of the summary nodes with role `LN` (shown as `LN`-labelled rectangles in Figure 6-11). Note that the sets of objects mapped to these two summary nodes are disjoint. The first summary `LN`-node represents objects stored in the list before the object referenced by `prev`. The second summary `LN`-node represents objects stored in the list after the object referenced by `current`. $\triangle$

6.6.2 Transfer Functions

The key complication in developing the transfer functions for the role analysis is to accurately model the movement of objects onstage and offstage. For example, a load statement `x=y.f` may cause the object referred to by `y.f` to move onstage. In addition, if `x` was the only reference to an onstage object o before the statement executed, object o moves offstage after the execution of the load statement, and thus must satisfy the $\mathsf{locallyConsistent}$ predicate.

The analysis uses an expansion relation $\preceq$ to model the movement of objects onstage and a contraction relation $\succeq$ to model the movement of objects offstage. The expansion relation uses the invariant that offstage nodes have correct roles to generate possible aliasing relationships for the node being pulled onstage. The contraction relation establishes the role invariants for the node going offstage, allowing the node to be merged into the other offstage nodes and represented more compactly.

We present our role analysis as an abstract execution relation $\overset{\mathtt{st}}{\rightsquigarrow}$. The abstract execution ensures that the abstraction relation α is a forward simulation relation [150] from the space of concrete heaps with role assignments to the set $\mathsf{RoleGraphs}$. The simulation relation implies that the traces of $\rightsquigarrow$ include the traces of the instrumented semantics $\rightarrow$. To ensure that the program does not violate constraints associated with roles, it is thus sufficient to guarantee that $\perp_G$ is not reachable via $\rightsquigarrow$.

To prove that $\perp_G$ is not reachable in the abstract execution, the analysis computes for every program point p a set of role graphs $\mathcal{G}$ that conservatively approximates the possible program states at point p. The transfer function for a statement st is an image $[\![\mathsf{st}]\!](\mathcal{G}) = \{G' \mid G \in \mathcal{G}, G \overset{\mathtt{st}}{\rightsquigarrow} G'\}$. The analysis computes the relation $\overset{\mathtt{st}}{\rightsquigarrow}$ in three steps:

$$\begin{tikzcd}
 & \langle H_c, \rho_c \rangle \arrow[r] \arrow[dl, "\alpha"'] \arrow[d, "\alpha"'] & \langle H'_c, \rho'_c \rangle \arrow[dr, "\alpha"] & \\
G_1 \arrow[r, phantom, "\preceq"] & G_2 \arrow[r, phantom, "\overset{\texttt{st}}{\Longrightarrow}"] & G_3 \arrow[r, phantom, "\succeq"] & G_4
\end{tikzcd}$$

Figure 6-12: Simulation Relation Between Abstract and Concrete Execution

Transition	Definition	Conditions
$\langle H, \rho, K\rangle \overset{\texttt{x=y.f}}{\rightsquigarrow} G'$	$\langle H, \rho, K\rangle \overset{n_y,f}{\preceq} G_1 \overset{\texttt{x=y.f}}{\Longrightarrow} G_2 \overset{n_x}{\succeq} G'$	$\langle \mathsf{proc}, \mathtt{x}, n_x\rangle, \langle \mathsf{proc}, \mathtt{y}, n_y\rangle \in H$
$\langle H, \rho, K\rangle \overset{\texttt{x=y}}{\rightsquigarrow} G'$	$\langle H, \rho, K\rangle \overset{\texttt{x=y}}{\Longrightarrow} G_1 \overset{n_1}{\succeq} G'$	$\langle \mathsf{proc}, \mathtt{x}, n_1\rangle \in H$
$\langle H, \rho, K\rangle \overset{\texttt{x=new}}{\rightsquigarrow} G'$	$\langle H, \rho, K\rangle \overset{\texttt{x=new}}{\Longrightarrow} G_1 \overset{n_1}{\succeq} G'$	$\langle \mathsf{proc}, \mathtt{x}, n_1\rangle \in H$
$\langle H, \rho, K\rangle \overset{\texttt{st}}{\rightsquigarrow} G'$	$\langle H, \rho, K\rangle \overset{\texttt{st}}{\Longrightarrow} G'$	$\mathtt{st} \in \{\texttt{x.f=y},$ $\texttt{test(c)},$ $\texttt{setRole(x:r)},$ $\texttt{roleCheck(}x_{1..p}\texttt{, ra)}\}$

Figure 6-13: Abstract Execution $\rightsquigarrow$

1. ensure that the relevant nodes are instantiated using expansion relation $\preceq$ (Section 6.6.2);
2. perform symbolic execution $\overset{\texttt{st}}{\Longrightarrow}$ of the statement `st` (Section 6.6.2);
3. merge nodes if needed using contraction relation $\succeq$ to keep the role graph bounded (Section 6.6.2).

Figure 6-12 shows how the abstraction relation α relates $\preceq$, $\overset{\texttt{st}}{\Longrightarrow}$, and $\succeq$ with the concrete execution $\rightarrow$ in instrumented semantics. Assume that a concrete heap $\langle H_c, \rho_c\rangle$ is represented by the role graph G_1. Then one of the role graphs G_2 obtained after expansion remains an abstraction of $\langle H_c, \rho_c\rangle$. The symbolic execution $\overset{\texttt{st}}{\Longrightarrow}$ followed by the contraction relation $\succeq$ corresponds to the instrumented operational semantics $\rightarrow$.

Figure 6-13 shows rules for the abstract execution relation $\overset{\texttt{st}}{\rightsquigarrow}$. Only Load statement uses the expansion relation, because the other statements operate on objects that are already onstage. Load, Copy, and New statements may remove a local variable reference from an object, so they use contraction relation to move the object offstage if needed. For the rest of the statements, the abstract execution reduces to symbolic execution $\Longrightarrow$ described in Section 6.6.2.

Nondeterminism and Failure The $\overset{\texttt{st}}{\rightsquigarrow}$ relation is not a function because the expansion relation $\preceq$ can generate a set of role graphs from a single role graph. Also,

Transition	Definition	Condition
$\langle H,\rho,K\rangle \overset{n,f}{\preceq} \langle H,\rho,K\rangle$		$\langle n,f,n'\rangle \in H, n' \in \mathsf{onstage}(H)$
$\langle H,\rho,K\rangle \overset{n,f}{\preceq} G'$	$\langle H,\rho,K\rangle \underset{n'}{\overset{n_0}{\Uparrow}} \langle H_1,\rho_1,K_1\rangle \overset{n_0}{\parallel} G'$	$\langle n,f,n'\rangle \in H, n' \in \mathsf{offstage}(H)$ $\langle n,f,n_0\rangle \in H_1$

Figure 6-14: Expansion Relation

$\langle H,\rho,K\rangle \underset{n'}{\overset{n_0}{\Uparrow}} \langle H',\rho',K'\rangle$	$H' = H \setminus H_0 \cup H_0' \cup H_1'$ $\rho' = \rho[n_0 \mapsto \rho(n')]$ $K' = K[n_0 \mapsto i]$ $\mathsf{localCheck}(n_0, \langle H',\rho',K'\rangle)$ $H_0 \subseteq H \cap \big(\mathsf{onstage}(H) \times F \times \{n'\} \cup \{n'\} \times F \times \mathsf{onstage}(H)\big)$ $H_1 \subseteq H \cap \big(\mathsf{offstage}(H) \times F \times \{n'\} \cup \{n'\} \times F \times \mathsf{offstage}(H)\big)$ $H_0' = \mathsf{swing}(n', n_0, H_0)$ $H_1' \subseteq \mathsf{swing}(n', n_0, H_1)$

$$\begin{aligned}\mathsf{swing}(n_{\mathsf{old}}, n_{\mathsf{new}}, H) \quad = \quad & \{\langle n_{\mathsf{new}}, f, n\rangle \mid \langle n_{\mathsf{old}}, f, n\rangle \in H\} \cup \\ & \{\langle n, f, n_{\mathsf{new}}\rangle \mid \langle n, f, n_{\mathsf{old}}\rangle \in H\} \cup \\ & \{\langle n_{\mathsf{new}}, f, n_{\mathsf{new}}\rangle \mid \langle n_{\mathsf{old}}, f, n_{\mathsf{old}}\rangle \in H\}\end{aligned}$$

Figure 6-15: Instantiation Relation

there might be no $\overset{\mathtt{st}}{\leadsto}$ transitions originating from a given state G if the symbolic execution $\Longrightarrow$ produces no results. This corresponds to a trace which cannot be extended further due to a `test` statement which fails in state G. This is in contrast to a transition from G to $\perp_G$ which indicates a potential role consistency violation or a null pointer dereference. We assume that $\Longrightarrow$ and $\succeq$ relations contain the transition $\langle \perp_G, \perp_G\rangle$ to propagate the error role graph. In most cases we do not show the explicit transitions to error states.

Expansion

Figure 6-14 shows the expansion relation $\overset{n,f}{\preceq}$. Given a role graph $\langle H,\rho,K\rangle$, expansion attempts to produce a set of role graphs $\langle H',\rho',K'\rangle$ in each of which $\langle n,f,n_0\rangle \in H'$ and $K(n_0) = i$. Expansion is used in abstract execution of the Load statement. It first checks for null pointer dereference and reports an error if the check fails. If $\langle n,f,n'\rangle \in H$ and $K(n') = i$ already hold, the expansion returns the original state. Otherwise, $\langle n,f,n'\rangle \in H$ with $K(n') = s$. In that case, the summary node n' is first instantiated using instantiation relation $\underset{n'}{\overset{n_0}{\Uparrow}}$. Next, the split relation $\overset{n_0}{\parallel}$ is applied. Let $\rho(n_0) = r$. The split relation ensures that n_0 is not a member of any cycle of offstage nodes which contains only edges in $\mathsf{acyclic}(r)$. We explain instantiation and split in more detail below.

Instantiation Figure 6-15 presents the instantiation relation. Given a role graph $G = \langle H, \rho, K \rangle$, instantiation $\overset{n_0}{\underset{n'}{\Uparrow}}$ generates the set of role graphs $\langle H', \rho', K' \rangle$ such that each concrete heap represented by $\langle H, \rho, K \rangle$ is represented by one of the graphs $\langle H', \rho', K' \rangle$. Each of the new role graphs contains a fresh individual node n_0 that satisfies $\mathsf{localCheck}$. The edges of n_0 are a subset of edges from and to n'.

Let H_0 be a subset of the references between n' and onstage nodes, and let H_1 be a subset of the references between n' and offstage nodes. References in H_0 are moved from n' to the new node n_0, because they represent at most one reference, while references in H_1 are copied to n_0 because they may represent multiple concrete heap references. Moving a reference is formalized via the swing operation in Figure 6-15.

The instantiation of a single graph can generate multiple role graphs depending on the choice of H_0' and H_1'. The number of graphs generated is limited by the existing references of node n' and by the $\mathsf{localCheck}$ requirement for n_0. This is where our role analysis takes advantage of the constraints associated with role definitions to reduce the number of aliasing possibilities that need to be considered.

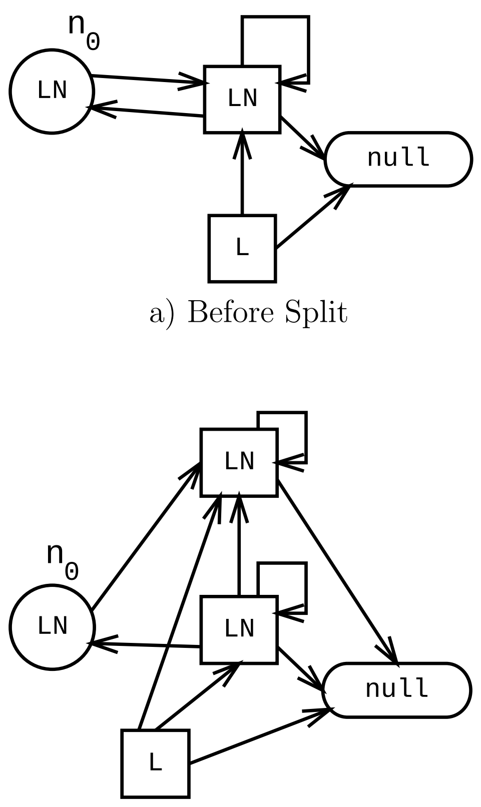
Split The split relation is important for verifying operations on data structures such as skip lists and sparse matrices. It is also useful for improving the precision of the initial set of role graphs on procedure entry (Section 6.7.2).

The goal of the split relation is to exploit the acyclicity constraints associated with role definitions. After a node n_0 is brought onstage, split represents the acyclicity condition of $\rho(n_0)$ explicitly by eliminating impossible paths in the role graph. It uses additional offstage nodes to encode the reachability information implied by the acyclicity conditions. This information can then be used even after the role of node n_0 changes. In particular, it allows the acyclicity condition of n_0 to be verified when n_0 moves offstage.

Example 24 Consider a role graph for an acyclic list with nodes `LN` and a header node `L`. The instantiated node n_0 is in the middle of the list. Figure 6-16 a) shows a role graph with a single summary node representing all offstage `LN`-nodes. Figure 6-16 b) shows the role graph after applying the split relation. The resulting role graph contains two `LN` summary nodes. The first `LN` summary node represents objects definitely reachable from n_0 along `next` edges; the second summary `NL` node represents objects definitely not reachable from n_0. $\triangle$

Figure 6-17 shows the definition of the split operation on node n_0, denoted by $\overset{n_0}{\|}$. Let $G = \langle H, \rho, K \rangle$ be the initial role graph and $\rho(n_0) = r$. If $\mathsf{acyclic}(r) = \emptyset$, then the split operation returns the original graph G; otherwise it proceeds as follows. Call a path in graph H *cycle-inducing* if all of its nodes are offstage and all of its edges are in $\mathsf{acyclic}(r)$. Let S_{cyc} be the set of nodes n such that there is a cycle-inducing path from n_0 to n and a cycle-inducing path from n to n_0.

The goal of the split operation is to split the set S_{cyc} into a fresh set of nodes S_{NR} representing objects definitely not reachable from n_0 along edges in $\mathsf{acyclic}(r)$ and a fresh set of nodes S_{R} representing objects definitely reachable from n_0. Each of the newly generated graphs H' has the following properties:



a) Before Split

b) After Split

Figure 6-16: A Role Graph for an Acyclic List

$$\langle H, \rho, K\rangle \overset{n_0}{\parallel} \langle H, \rho, K\rangle, \quad \mathsf{acycCheck}(n_0, \langle H, \rho, K\rangle, \mathsf{offstage}(H))$$
$$\langle H, \rho, K\rangle \overset{n_0}{\parallel} \langle H', \rho', K'\rangle, \quad \neg\mathsf{acycCheck}(n_0, \langle H, \rho, K\rangle, \mathsf{offstage}(H))$$

where

$$\begin{aligned}
&H' = (H \setminus H_{\text{cyc}}) \cup H_{\text{off}} \cup B_{\text{fNR}} \cup B_{\text{fR}} \cup B_{\text{tNR}} \cup B_{\text{tR}} \cup N_{\text{f}} \cup N_{\text{t}}\\
&H_{\text{cyc}} = \{\langle n_1, f, n_2\rangle \mid n_1 \text{ or } n_2 \in S_{\text{cyc}}\}\\
&H_{\text{off}} \;=\; \Big\{ \langle n_1', f, n_2'\rangle \;\mid\; n_1 = c(n_1'), n_2 = c(n_2'),\\
&\qquad\qquad\qquad\qquad n_1, n_2 \in \mathsf{offstage}_1(H), n_1 \text{ or } n_2 \in S_{\text{cyc}},\\
&\qquad\qquad\qquad\qquad \langle n_1, f, n_2\rangle \in H \Big\}\\
&\qquad\quad \setminus (S_{\text{R}} \times \mathsf{acyclic}(r) \times S_{\text{NR}})\\
&H \cap (\mathsf{onstage}(H) \times F \cup \{n_0\} \times \mathsf{acyclic}(r)) \times S_{\text{cyc}} = A_{\text{fNR}} \uplus A_{\text{fR}}\\
&H \cap S_{\text{cyc}} \times (\mathsf{acyclic}(r) \times \{n_0\} \cup F \times \mathsf{onstage}(H)) = A_{\text{tNR}} \uplus A_{\text{tR}}\\
&B_{\text{fNR}} = \{\langle n_1, f, h_{\text{NR}}(n_2)\rangle \mid \langle n_1, f, n_2\rangle \in A_{\text{fNR}}\}\\
&B_{\text{fR}} = \{\langle n_1, f, h_{\text{R}}(n_2)\rangle \mid \langle n_1, f, n_2\rangle \in A_{\text{fR}}\}\\
&B_{\text{tNR}} = \{\langle h_{\text{NR}}(n_1), f, n_2\rangle \mid \langle n_1, f, n_2\rangle \in A_{\text{tNR}}\}\\
&B_{\text{tR}} = \{\langle h_{\text{R}}(n_1), f, n_2\rangle \mid \langle n_1, f, n_2\rangle \in A_{\text{tR}}\}\\
&N_{\text{f}} = \{\langle n_0, f, n'\rangle \mid n' \in S_{\text{R}}, \langle n_0, f, c(n')\rangle \in H, f \in \mathsf{acyclic}(r)\}\\
&N_{\text{t}} = \{\langle n', f, n_0\rangle \mid n' \in S_{\text{NR}}, \langle c(n'), f, n_0\rangle \in H, f \in \mathsf{acyclic}(r)\}\\
&S_{\text{cyc}} = \{n \mid \exists n_1, \ldots, n_{p-1} \in \mathsf{offstage}(H) :\\
&\qquad \langle n_0, f_0, n_1\rangle, \ldots, \langle n_k, f_k, n\rangle, \langle n, f_{k+1}, n_{k+2}\rangle, \langle n_{p-1}, f_{p-1}, n_0\rangle \in H,\\
&\qquad f_0, \ldots, f_{p-1} \in \mathsf{acyclic}(r)\}\\
&\mathsf{offstage}_1(H) = \mathsf{offstage}(H) \setminus \{n_0\}\\
&r = \rho(n_0)\\
&\\
&\rho'(c(n)) = \rho(n)\\
&K'(c(n)) = K(n)
\end{aligned}$$

Figure 6-17: Split Relation

1) merging the corresponding nodes from S_{NR} and S_{R} in H' yields the original graph H;

2) n_0 is not a member of any cycle in H' consisting of offstage nodes and edges in $\mathsf{acyclic}(r)$;

3) onstage nodes in H' have the same number of fields and aliases as in H.

Let $S_0 = \mathsf{nodes}(H) \setminus S_{\mathrm{cyc}}$ and let $h_{\mathrm{NR}} : S_{\mathrm{cyc}} \to S_{\mathrm{NR}}$ and $h_{\mathrm{R}} : S_{\mathrm{cyc}} \to S_{\mathrm{R}}$ be bijections. Define a function $c : \mathsf{nodes}(H') \to \mathsf{nodes}(H)$ as follows:

$$c(n) = \begin{cases} n, & n \in S_0 \\ h_{\mathrm{R}}^{-1}(n), & n \in S_{\mathrm{R}} \\ h_{\mathrm{NR}}^{-1}(n), & n \in S_{\mathrm{NR}} \end{cases}$$

Then $H' \subseteq \{\langle n_1', f, n_2' \rangle \mid \langle c(n_1'), f, c(n_2') \rangle \in H\}$.

Because there are two copies of S_0 in H', there might be multiple edges $\langle n_1', f, n_2' \rangle$ in H' corresponding to an edge $\langle c(n_1), f, c(n_2) \rangle \in H$.

If both n_1' and n_2' are offstage nodes other than n_0, we always include $\langle n_1', f, n_2' \rangle$ in H' unless $\langle n_1', f, n_2' \rangle \in S_{\mathrm{R}} \times \mathsf{acyclic}(r) \times S_{\mathrm{NR}}$. The last restriction prevents cycles in H'.

For an edge $\langle n_1, f, n_2 \rangle \in H$ where $n_1 \in \mathsf{onstage}(H)$ and $n_2 \in S_{\mathrm{cyc}}$ we include in H' either the edge $\langle n_1, f, h_{\mathrm{NR}}(n_2) \rangle$ or $\langle n_1, f, h_{\mathrm{R}}(n_2) \rangle$ but not both. Split generates multiple graphs H' to cover both cases. We proceed analogously if $n_2 \in \mathsf{onstage}(H)$ and $n_1 \in S_{\mathrm{cyc}}$. The node n_0 itself is treated in the same way as onstage nodes for $f \notin \mathsf{acyclic}(r)$. If $f \in \mathsf{acyclic}(r)$ then we choose references *to* n_0 to have a source in S_{NR}, whereas the reference *from* n_0 have the target in S_{R}.

Details of the split construction are given in Figure 6-17. The intuitive meaning of the sets of edges is the following:

H_{off}	: edges between offstage nodes
B_{fNR}	: edges from onstage nodes to S_{NR}
B_{fR}	: edges from onstage nodes to S_{R}
B_{tNR}	: edges from S_{NR} to onstage nodes
B_{tR}	: edges from S_{R} to onstage nodes
N_{f}	: $\mathsf{acyclic}(r)$-edges from n_0 to S_{R}
N_{t}	: $\mathsf{acyclic}(r)$-edges from S_{NR} to n_0

The sets B_{fNR} and B_{fR} are created as images of the sets A_{fNR} and A_{fR} which partition edges from onstage nodes to nodes in S_{cyc}. Similarly, the sets B_{tNR} and B_{tR} are created as images of the sets A_{tNR} and A_{tR} which partition edges from nodes in S_{cyc} to onstage nodes.

We note that if in the split operation $S_{\mathrm{cyc}} = \emptyset$ then split has no effect and need not be performed. In Figure 6-16, after performing a single split, there is no need to split for subsequent elements of the list. Examples like this indicate that split will not be invoked frequently during the analysis.

$\langle H,\rho,K\rangle \overset{n}{\succeq} \langle H,\rho,K\rangle$	$\exists \mathsf{x} \in \mathsf{var}(\mathsf{proc}) :$ $\langle \mathsf{proc}, \mathsf{x}, n\rangle \in H$
$\langle H,\rho,K\rangle \overset{n}{\succeq} \mathsf{normalize}(\langle H,\rho,K\rangle)$	$\mathsf{nodeCheck}(n, \langle H,\rho,K\rangle, \mathsf{offstage}(H))$

Figure 6-18: Contraction Relation

$$\mathsf{normalize}(\langle H,\rho,K\rangle) = \langle H',\rho',K'\rangle$$

$$\begin{aligned}
\text{where}\quad & H' = \{\langle n_{1/\sim}, f, n_{2/\sim}\rangle \mid \langle n_1, f, n_2\rangle \in H\} \\
& \rho'(n_{/\sim}) = \rho(n) \\
& K'(n_{/\sim}) = \begin{cases} i, & n_{/\sim} = \{n\}, K(n) = i \\ s, & \text{otherwise} \end{cases} \\
& n_1 \sim n_2 \text{ iff } n_1 = n_2 \text{ or} \\
& \qquad (n_1, n_2 \in \mathsf{offstage}(H), \rho(n_1) = \rho(n_2), \\
& \qquad \forall n_0 \in \mathsf{onstage}(H) : (\mathsf{reach}(n_0, n_1) \text{ iff } \mathsf{reach}(n_0, n_2)) \\
& \mathsf{reach}(n_0, n) \text{ iff } \exists n_1, \ldots, n_{p-1} \in \mathsf{offstage}(n), \exists f_1, \ldots, f_p \in \mathsf{acyclic}(\rho(n_0)) : \\
& \qquad \langle n_0, f_1, n_1\rangle, \ldots, \langle n_{p-1}, f_p, n\rangle \in H
\end{aligned}$$

Figure 6-19: Normalization

Contraction

Figure 6-18 shows the non-error transitions of the contraction relation $\overset{n}{\succeq}$. The analysis uses contraction when a reference to node n is removed. If there are other references to n, the result is the original graph. Otherwise n has just gone offstage, so the analysis invokes nodeCheck. If the check fails, the result is $\perp_G$. If the role check succeeds, the contraction invokes normalization operation to ensure that the role graph remains bounded. For simplicity, we use normalization whenever nodeCheck succeeds, although it is sufficient to perform normalization only at program points adjacent to back edges of the control-flow graph.

Normalization Figure 6-19 shows the normalization relation. Normalization accepts a role graph $\langle H,\rho,K\rangle$ and produces a normalized role graph $\langle H',\rho',K'\rangle$ which is a factor graph of $\langle H,\rho,K\rangle$ under the equivalence relation $\sim$. Two offstage nodes are equivalent under $\sim$ if they have the same role and the same reachability from onstage nodes. Here we consider node n to be reachable from an onstage node n_0 iff there is some path from n_0 to n whose edges belong to acyclic($\rho(n_0)$) and whose nodes are all in offstage(H). Note that, by construction, normalization avoids merging nodes which were previously generated in the split operation $\|$, while still ensuring a bound on the size of the role graph. For a procedure with l local variables, f fields and r roles the number of nodes in a role graph is on the order of $r2^l$ so the maximum size of a chain in the lattice is of the order of 2^{r2^l}. To ensure termination we consider role graphs equal up to isomorphism. Isomorphism checking can be done efficiently if normalization assigns canonical names to the equivalence classes it creates.

Statement s	Transition	Conditions
`x = y.f`	$\langle H \uplus \{\mathsf{proc}, \mathsf{x}, n_x\}, \rho, K\rangle \overset{\mathtt{st}}{\Longrightarrow} \langle H \uplus \{\mathsf{proc}, \mathsf{x}, n_f\}, \rho, K\rangle$	$\langle \mathsf{proc}, \mathsf{y}, n_y\rangle, \langle n_y, f, n_f\rangle \in H$
`x.f = y`	$\langle H \uplus \{n_x, f, n_f\}, \rho, K\rangle \overset{\mathtt{st}}{\Longrightarrow} \langle H \uplus \{n_x, f, n_y\}, \rho, K\rangle$	$\langle \mathsf{proc}, \mathsf{x}, n_x\rangle, \langle \mathsf{proc}, \mathsf{y}, n_y\rangle \in H$ $n_f \in \mathsf{onstage}(H)$
`x = y`	$\langle H \uplus \{\mathsf{proc}, \mathsf{x}, n_x\}, \rho, K\rangle \overset{\mathtt{st}}{\Longrightarrow} \langle H \uplus \{\mathsf{proc}, \mathsf{x}, n_y\}, \rho, K\rangle$	$\langle \mathsf{proc}, \mathsf{y}, n_y\rangle \in H$
`x = new`	$\langle H \uplus \{\mathsf{proc}, \mathsf{x}, n_x\}, \rho, K\rangle \overset{\mathtt{st}}{\Longrightarrow} \langle H \uplus \{\mathsf{proc}, \mathsf{x}, n_n\}, \rho', K\rangle$	n_n fresh $\rho' = \rho[n_n \mapsto \mathsf{unknown}]$
`test(c)`	$\langle H, \rho, K\rangle \overset{\mathtt{st}}{\Longrightarrow} \langle H, \rho, K\rangle$	$\mathsf{satisfied}(\mathsf{c}, H)$
`setRole(x:r)`	$\langle H, \rho, K\rangle \overset{\mathtt{st}}{\Longrightarrow} \langle H, \rho[n_x \mapsto \mathtt{r}], K\rangle$	$\langle \mathsf{proc}, \mathsf{x}, n_x\rangle \in H$ $\mathsf{roleChOk}(n_x, \mathtt{r}, \langle H, \rho, K\rangle)$
`roleCheck(`$x_{1..p}$, ra`)`	$\langle H, \rho, K\rangle \overset{\mathtt{st}}{\Longrightarrow} \langle H, \rho, K\rangle$	$\forall i \; \langle \mathsf{proc}, \mathsf{x}_i, n_i\rangle \in H$ $\mathsf{nodeCheck}(n_i, \langle H, \rho, K\rangle, S)$ $S = \mathsf{offstage}(H) \cup \{n_i\}_i$ $\rho(n_i) = \mathsf{ra}(n_i)$

$$\mathsf{satisfied}(\mathsf{x{=}{=}y}, H_c) \text{ iff } \{o \mid \langle \mathsf{proc}, \mathsf{x}, o\rangle \in H_c\} = \{o \mid \langle \mathsf{proc}, \mathsf{y}, o\rangle \in H_c\}$$

$$\mathsf{satisfied}(\mathsf{!(x{=}{=}y)}, H_c) \text{ iff not } \mathsf{satisfied}(\mathsf{x{=}{=}y}, H_c)$$

Figure 6-20: Symbolic Execution of Basic Statements

Symbolic Execution

Figure 6-20 shows the symbolic execution relation $\overset{\mathtt{st}}{\Longrightarrow}$. In most cases, the symbolic execution of a statement acts on the abstract heap in the same way that the statement would act on the concrete heap. In particular, the Store statement always performs strong updates. The simplicity of symbolic execution is due to conditions 3) and 5) in the abstraction relation α. These conditions are ensured by the $\preceq$ relation which instantiates nodes, allowing strong updates. The symbolic execution also verifies the consistency conditions that are not verified by $\preceq$ or $\succeq$.

Verifying Reference Removal Consistency The abstract execution $\overset{\mathtt{st}}{\rightsquigarrow}$ for the Store statement can easily verify the Store safety condition from section 6.5.4, because the set of onstage and offstage nodes is known precisely for every role graph. It returns $\perp_G$ if the safety condition fails.

Symbolic Execution of setRole The `setRole(x:r)` statement sets the role of node n_x referenced by variable `x` to `r`. Let $G = \langle H, \rho, K\rangle$ be the current role graph and let $\langle \mathsf{proc}, \mathtt{x}, n_x\rangle \in H$. If n_x has no adjacent offstage nodes, the role change always succeeds. In general, there are restrictions on when the change can be done. Let $\langle H_c, \rho_c\rangle$ be a concrete heap with role assignment represented by G and h be a homomorphism from H_c to H. Let $h(o_x) = n_x$. Let $r_0 = \rho_c(o_x)$. The symbolic execution must make sure that the condition $\mathsf{conW}(\rho_c, H_c, \mathsf{offstage}(H_c))$ continues to hold after the role change. Because the set of onstage nodes does not change, it suffices to ensure that the original roles for offstage nodes are consistent with the new role r. The acyclicity constraint involves only offstage nodes, so it remains satisfied. The other role constraints are local, so they can only be violated for offstage neighbors of n_x. To make sure that no violations occur, we require:

1. $\mathtt{r} \in \mathsf{field}_f(\rho(n))$ for all $\langle n, f, n_x\rangle \in H$, and

2. $\langle \mathtt{r}, f\rangle \in \mathsf{slot}_i(\rho(n))$ for all $\langle n_x, f, n\rangle \in H$ and every slot i such that $\langle r_0, f\rangle \in \mathsf{slot}_i(\rho(n))$

This is sufficient to guarantee $\mathsf{conW}(\rho_c, H_c, \mathsf{offstage}(H_c))$. To ensure condition 2) in Definition 22 of the abstraction relation, we require that for every $\langle f, g\rangle \in \mathsf{identities}(\mathtt{r})$,

1. $\langle f, g\rangle \in \mathsf{identities}(r_0)$ or
2. for all $\langle n_x, f, n\rangle \in H$: $K(n) = i$ and $(\langle n, g, n'\rangle \in H$ implies $n' = n_x)$.

Symbolic Execution of roleCheck The symbolic execution of the statement $\mathtt{roleCheck}(x_1, \dots, x_p, \mathsf{ra})$ ensures that the conW predicate of the concrete semantics is satisfied for the concrete heaps which correspond to the current abstract role graph. The symbolic execution returns the error graph $\bot_G$ if ρ is inconsistent with ra or if any of the nodes n_i referenced by x_i fail to satisfy nodeCheck.

Accessibility Condition The analysis ensures that the accessibility condition for the Load statement will be satisfied in procedure proc before procedure proc is called. This technique makes use of procedure effects and is described in Section 6.7.

Node Check

The analysis uses the nodeCheck predicate to incrementally maintain the abstraction relation. We first define the predicate localCheck, which roughly corresponds to the predicate locallyConsistent (Definition 2), but ignores the nonlocal acyclicity condition and additionally ensures condition 2) from Definition 22.

Definition 25 *For a role graph $G = \langle H, \rho, K\rangle$, an individual node n and a set S, the predicate $\mathsf{localCheck}(n, G)$ holds iff the following conditions are met. Let $r = \rho(n)$.*

1A. (Outgoing fields check) For fields $f \in F$, if $\langle n, f, n'\rangle \in H$ then $\rho(n') \in \mathsf{field}_f(r)$.

2A. (Incoming slots check) Let $\{\langle n_1, f_1\rangle, \dots, \langle n_k, f_k\rangle\} = \{\langle n', f\rangle \mid \langle n', f, n\rangle \in H\}$ be the set of all aliases of node n in abstract heap H. Then $k = \mathsf{slotno}(r)$ and there exists a permutation p of the set $\{1, \dots, k\}$ such that $\langle \rho(n_i), f_i\rangle \in \mathsf{slot}_{p_i}(r)$ for all i.

3A. (Identity Check) If $\langle n, f, n'\rangle \in H$, $\langle n', g, n''\rangle \in H$, $\langle f, g\rangle \in \mathsf{identities}(r)$, and $K(n') = i$, then $n = n''$.

4A. (Neighbor Identity Check) For every edge $\langle n', f, n\rangle \in H$, if $K(n') = i$, $\rho(n') = r'$ and $\langle f, g\rangle \in \mathsf{identities}(r')$ then $\langle n, g, n'\rangle \in H$.

5A. (Field Sanity Check) For every $f \in F$ there is exactly one edge $\langle n, f, n'\rangle \in H$.

Conditions 1A and 2A correspond to conditions 1) and 2) in Definition 2. Condition 3) in Definition 19 is not necessarily implied by condition 3A) if some of the neighbors of n are summary nodes. Condition 3) cannot be established based only on summary nodes, because verifying an identity constraint for field f of node n where $\langle n, f, n'\rangle \in$

H requires knowing the identity of n', not only its existence and role. We therefore rely on Condition 2) of the Definition 22 to ensure that identity relations of neighbors of node n are satisfied before n moves offstage.

The predicate $\mathsf{acycCheck}(n, G, S)$ verifies the acyclicity condition from Definition 19.

Definition 26 *We say that node* $n \in \mathsf{nodes}(H)$ *satisfies an acyclicity check in graph* $G = \langle H, \rho, K \rangle$ *with respect to set* S*, and we write* $\mathsf{acycCheck}(n, G, S)$*, iff it is not the case that* H *contains a cycle* $n_1, f_1, \ldots, n_s, f_s, n_1$ *where* $n_1 = n$*,* $f_1, \ldots, f_s \in \mathsf{acyclic}(\rho(n))$ *and* $n_1, \ldots, n_s \in S$*.*

This enables us to define the $\mathsf{nodeCheck}$ predicate.

Definition 27 $\mathsf{nodeCheck}(n, G, S)$ *holds iff both the predicate* $\mathsf{localCheck}(n, G)$ *and the predicate* $\mathsf{acycCheck}(n, G, S)$ *hold.*

6.7 Interprocedural Role Analysis

This section describes the interprocedural aspects of our role analysis. Interprocedural role analysis can be viewed as an instance of the functional approach to interprocedural data-flow analysis [181]. For each program point p, the role analysis approximates program traces from procedure entry to point p. The solution in [181] proposes tagging the entire data-flow fact G at point p with the data flow fact G_0 at procedure entry. In contrast, our analysis computes the correspondence between the heaps at procedure entry and the heaps at point p at the granularity of sets of objects that constitute the role graphs. This allows our analysis to detect which regions of the heap have been modified. We approximate the concrete executions of a procedure with *procedure transfer relations* consisting of 1) an initial context and 2) a set of *effects*. Effects are fine-grained transfer relations which summarize load and store statements and can naturally describe local heap modifications. In this work we assume that procedure transfer relations are supplied and we are concerned with a) verifying that transfer relations are a conservative approximation of procedure implementation b) instantiating transfer relations at call sites.

6.7.1 Procedure Transfer Relations

A transfer relation for a procedure proc extends the procedure signature with an initial context denoted $\mathsf{context}(\mathsf{proc})$, and procedure effects denoted $\mathsf{effect}(\mathsf{proc})$.

Initial Context

Figures 6-21 and 6-22 contain examples of initial context specification. An initial context is a description of the initial role graph $\langle H_{\mathsf{IC}}, \rho_{\mathsf{IC}}, K_{\mathsf{IC}} \rangle$ where ρ_{IC} and K_{IC} are determined by a `nodes` declaration and H_{IC} is determined by a `edges` declaration. The initial role graph specifies a set of concrete heaps at procedure entry and assigns names for sets of nodes in these heaps. The next definition is similar to Definition 22.

Definition 28 *We say that a concrete heap $\langle H_c, \rho_c \rangle$ is represented by the initial role graph $\langle H_{\mathsf{IC}}, \rho_{\mathsf{IC}}, K_{\mathsf{IC}} \rangle$ and write $\langle H_c, \rho_c \rangle\, \alpha_0 \langle H_{\mathsf{IC}}, \rho_{\mathsf{IC}}, K_{\mathsf{IC}} \rangle$, iff there exists a function $h_0 : \mathsf{nodes}(H_c) \to \mathsf{nodes}(H_{\mathsf{IC}})$ such that*

1. $\mathsf{conW}(\rho_c, H_c, h_0^{-1}(\mathsf{read}(\mathsf{proc}))$;
2. h_0 *is a graph homomorphism;*
3. $K_{\mathsf{IC}}(n) = i$ *implies* $|h_0^{-1}(n)| \leq 1$;
4. $h_0(\mathsf{null}_c) = \mathsf{null}$ *and* $h_0(\mathsf{proc}_c) = \mathsf{proc}$;
5. $\rho_c(o) = \rho_{\mathsf{IC}}(h_0(o))$ *for every object* $o \in \mathsf{nodes}(H_c)$.

Here $\mathsf{read}(\mathsf{proc})$ is the set of initial-context nodes read by the procedure (see below). For simplicity, we assume one context per procedure; it is straightforward to generalize the treatment to multiple contexts.

A context is specified by declaring a list of nodes and a list of edges.

A list of nodes is given with `nodes` declaration. It specifies a role for every node at procedure entry. Individual nodes are denoted with lowercase identifiers, summary nodes with uppercase identifiers. By using summary nodes it is possible to indicate disjointness of entire heap regions and reachability between nodes in the heap.

There are two kinds of edges in the initial role graph: parameter edges and heap edges. A parameter edge `p->pn` is interpreted as $\langle \mathsf{proc}, \mathtt{p}, \mathtt{pn} \rangle \in H_{\mathsf{IC}}$. We require every parameter edge to have an individual node as a target, we call such node a *parameter node*. The role of a parameter node referenced by $\mathsf{param}_i(\mathsf{proc})$ is always $\mathsf{preR}_i(\mathsf{proc})$. Since different nodes in the initial role graph denote disjoint sets of concrete objects, parameter edges

```
p1 -> n1
p2 -> n1
```

imply that parameters `p1` and `p2` must be aliased,

```
p1 -> n1
p2 -> n2
```

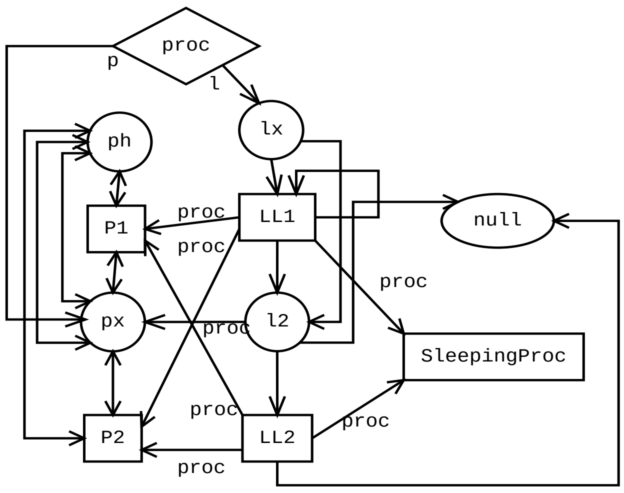
force `p1` and `p2` to be unaliased, whereas

```
p1 -> n1|n2
p2 -> n1|n2
```

allow for both possibilities. A heap edge `n -f-> m` denotes $\langle \mathtt{n}, \mathtt{f}, \mathtt{m} \rangle \in H_{\mathsf{IC}}$. The shorthand notation

```
n1 -f-> n2
   -g-> n3
```

denotes two heap edges $\langle \mathtt{n1}, \mathtt{f}, \mathtt{n2} \rangle, \langle \mathtt{n1}, \mathtt{g}, \mathtt{n3} \rangle \in H_{\mathsf{IC}}$. An expression `n1 -f-> n2|n3` denotes two edges `n1 -f-> n2` and `n1 -f-> n3`. We use similar shorthands for parameter edges.



```
nodes ph : RunningHeader,
      P1, px, P2 : RunningProc,
      lx : LiveHeader,
      LL1, l2, LL2 : LiveList;
edges p-> px, l-> px,
      ph -next-> P1|px
         -prev-> px|P2,
      P1 -next-> P1|px
         -prev-> ph|P1,
      px -next-> P2|ph
         -prev-> P1|ph,
      P2 -next-> P2|ph
         -prev-> P2|px,
      lx -next-> LL1|l2,
      LL1 -next-> LL1|l2
          -proc-> P1|P2|SleepingProc
      l2 -next-> LL2|null
         -proc-> px,
      LL2 -next-> LL2|null
          -proc-> P1|P2|SleepingProc
```

Figure 6-21: Initial Context for `kill` Procedure

Example 29 Figure 6-21 shows an initial context graph for the `kill` procedure from Example 17. It is a refinement of the role reference diagram of Figure 6-1 as it gives description of the heap specific to the entry of `kill` procedure. The initial context makes explicit the fact that there is only one header node for the list of running processes (`ph`) and one header node for the list of all active processes (`lx`). More importantly, it shows that traversing the list of active processes reaches a node `l2` whose `proc` field references the parameter node `px`. This is sufficient for the analysis to conclude that there will be no null pointer dereferences in the `while` loop of `kill` procedure since `l2` is reached before null. △

We assume that the initial context always contains the role reference diagram RRD (Definition 8). Nodes from RRD are called *anonymous nodes* and are referred to via role name. This further reduces the size of initial context specifications by leveraging global role definitions. In Figure 6-21 there is no need to specify edges originating from `SleepingProc` or even mention the node `SleepingTree`, since role definitions alone contain enough information on this part of the heap to enable the analysis of the procedure.

Procedure Effects

Procedure effects conservatively approximate the region of the heap that the procedure accesses and indicate changes to the referencing relationships in that region. There are two kinds of effects: read effects and write effects.

A *read effect* specifies a set read(proc) of initial graph nodes accessed by the procedure. It is used to ensure that the accessibility condition in Section 6.5.4 is satisfied. If the set of nodes denoted by read(proc) is mapped to a node n which is onstage in the caller but is not an argument of the procedure call, a role check error is reported at the call site.

Write effects are used to modify caller's role graph to conservatively model the procedure call. A write effect $e_1.f = e_2$ approximates Store operations within a procedure. The expression e_1 denotes objects being written to, f denotes the field written, and e_2 denotes the set of objects which could be assigned to the field. Write effects are *may* effects by default, which means that the procedure is free not to perform them. It is possible to specify that a write effect *must* be performed by prefixing it with a "`!`" sign.

Example 30 In Figure 6-22, the `insert` procedure inserts an isolated cell into the end of an acyclic singly linked list. As a result, the role of the cell changes to `LN`. The initial context declares parameter nodes `ln` and `xn` (whose initial roles are deduced from roles of parameters), and mentions anonymous `LN` node from a default copy of the role reference diagram RRD. The code of the procedure is summarized with two write effects. The first write effect indicates that the procedure may perform zero or more Store operations to field `next` of nodes mapped to `ln` or `LN` in context(proc). The second write effect indicates that the execution of the procedure must perform a Store to the field `next` of `xn` node where the reference stored is either a node mapped onto anonymous `LN` node or `null`. △

```
procedure insert(l : L,
                 x : IsolatedN ->> LN)
nodes ln, xn;
edges l-> ln, x-> xn,
      ln -next-> LN|null;
effects ln|LN . next = xn,
        ! xn.next = LN|null;
local c, p;
{
  p = l;
  c = l.next;
  while (c!=null) {
    p = c;
    c = p.next;
  }
  p.next = x;
  x.next = c;
  setRole(x:LN);
}
```

Figure 6-22: Insert Procedure for Acyclic List

Effects also describe assignments that procedures perform on the newly created nodes. Here we adopt a simple solution of using a single summary node denoted NEW to represent all nodes created inside the procedure. We write $\mathsf{nodes}_0(H_{\mathsf{IC}})$ for the set $\mathsf{nodes}(H_{\mathsf{IC}}) \cup \{\mathsf{NEW}\}$.

Example 31 Procedure `insertSome` in Figure 6-23 is similar to procedure `insert` in Figure 6-22, except that the node inserted is created inside the procedure. It is therefore referred to in effects via generic summary node NEW. $\triangle$

We represent all may write effects as a set $\mathsf{mayWr}(\mathsf{proc})$ of triples $\langle n_j, f, n'_j \rangle$ where $n, n'_j \in \mathsf{nodes}_0(H_{\mathsf{IC}})$ and $f \in F$. We represent must write effects as a sequence $\mathsf{mustWr}_j(\mathsf{proc})$ of subsets of the set $K_{\mathsf{IC}}^{-1}(i) \times F \times \mathsf{nodes}_0(H_{\mathsf{IC}})$. Here $1 \leq j \leq \mathsf{mustWrNo}(\mathsf{proc})$.

To simplify the interpretation of the declared procedure effects in terms of concrete reads and writes, we require the union $\cup_i \mathsf{mustWr}_i(\mathsf{proc})$ to be disjoint from the set $\mathsf{mayWr}(\mathsf{proc})$. We also require the nodes $n_1, \ldots, n_k$ in a must write effect $n_1 | \cdots | n_k.f = e_2$ to be individual nodes. This allows strong updates when instantiating effects (Section 6.7.3).

Semantics of Procedure Effects

We now give precise meaning to procedure effects. Our definition is slightly complicated by the desire to capture the set of nodes that are actually read in an execution while still allowing a certain amount of observational equivalence for write effects.

```
procedure insertSome(l : L)
nodes ln;
edges l-> ln,
      ln -next-> LN|null;
effects ln|LN . next = NEW,
        NEW.next = LN|null;
aux c, p, x;
{
  p = l;
  c = l.next;
  while (c!=null) {
    p = c;
    c = p.next;
  }
  x = new;
  p.next = x;
  x.next = c;
  setRole(x:LN);
}
```

Figure 6-23: Insert Procedure with Object Allocation

The effects of procedure proc define a subset of permissible program traces in the following way. Consider a concrete heap H_c with role assignment ρ_c such that $\langle H_c, \rho_c\rangle\, \alpha_0 \langle H_{\mathsf{IC}}, \rho_{\mathsf{IC}}, K_{\mathsf{IC}}\rangle$ with graph homomorphism h_0 from Definition 28. Consider a trace T starting from a state with heap H_c and role assignment ρ_c. Extract the subsequence of all loads and stores in trace T. Replace Load `x=y.f` by concrete read `read` o_x where o_x is the concrete object referenced by `x` at the point of Load, and replace Store `x.f=y` by a concrete write $o_x.f = o_y$ where o_x is the object referenced by `x` and o_y object referenced by `y` at the point of Store. Let $p_1, \ldots, p_k$ be the sequence of all concrete read statements and $q_1, \ldots, q_k$ the sequence of all concrete write statements. We say that trace T starting at H_c conforms to the effects iff for all choices of h_0 the following conditions hold:

1. $h_0(o) \in \mathsf{read}(\mathsf{proc})$ for every p_i of the form `read` o
2. there exists a subsequence $q_{i_1}, \ldots, q_{i_t}$ of $q_1, \ldots, q_k$ such that
   (a) executing $q_{i_1}, \ldots, q_{i_t}$ on H_c yields the same result as executing the entire sequence $q_1, \ldots, q_k$
   (b) the sequence $q_{i_1}, \ldots, q_{i_t}$ implements write effects of procedure proc

A typical way to obtain a sequence $q_{i_1}, \ldots, q_{i_t}$ from the sequence $q_1, \ldots, q_k$ is to consider only the last write for each pair $\langle o_i, f\rangle$ of object and field.

We say that a sequence $q_{i_1}, \ldots, q_{i_t}$ implements write effects $\mathsf{mayWr}(\mathsf{proc})$ and $\mathsf{mustWr}_i(\mathsf{proc})$ for $1 \leq i \leq i_0$, $i_0 = \mathsf{mustWrNo}$ if and only if there exists an injection $s : \{1, \ldots, i_0\} \to \{i_1, \ldots, i_t\}$ such that

1. $\langle h'(o), f, h'(o') \rangle \in \mathsf{mustWr}_i(\mathsf{proc})$ for every concrete write $q_{s(i)}$ of the form $o.f = o'$, and
2. $\langle h'(o), f, h'(o') \rangle \in \mathsf{mayWr}(\mathsf{proc})$ for all concrete writes q_i of the form $o.f = o'$ for $i \in \{i_1, \ldots, i_t\} \setminus \{s(1), \ldots, s(i_0)\}$.

Here $h'(n) = h_0(n)$ for $n \in \mathsf{nodes}(H_c)$ where H_c is the initial concrete heap and $h'(n) = \mathsf{NEW}$ otherwise.

It is possible (although not very common) for a single concrete heap H_c to have multiple homomorphisms h_0 to the initial context H_{IC}. Note that in this case we require the trace T to conform to effects for *all* possible valid choices of h_0. This places the burden of multiple choices of h_0 on procedure transfer relation verification (Section 6.7.2) but in turn allows the context matching algorithm in Section 6.7.3 to select an arbitrary homomorphism between a caller's role graph and an initial context.

6.7.2 Verifying Procedure Transfer Relations

In this section we show how the analysis makes sure that a procedure conforms to its specification, expressed as an initial context with a list of effects. To verify procedure effects, we extend the analysis representation from Section 6.6.1. A non-error role graph is now a tuple $\langle H, \rho, K, \tau, E \rangle$ where:

1. $\tau : \mathsf{nodes}(H) \to \mathsf{nodes}_0(H_{\mathsf{IC}})$ is initial context transformation that assigns an initial context node $\tau(n) \in \mathsf{nodes}(H_{\mathsf{IC}})$ to every node n representing objects that existed prior to the procedure call, and assigns NEW to every node representing objects created during procedure activation;
2. $E \subseteq \cup_i \mathsf{mustWr}_i(\mathsf{proc})$ is a list of must write effects that procedure has performed so far.

The initial context transformation τ tracks how objects have moved since the beginning of procedure activation and is essential for verifying procedure effects which refer to initial context nodes.

We represent the list E of performed must effects as a partial map from the set $K_{\mathsf{IC}}^{-1}(i) \times F$ to $\mathsf{nodes}_0(H_{\mathsf{IC}})$. This allows the analysis to perform must effect folding by recording only the last must effect for every pair $\langle n, f \rangle$ of individual node n and field f.

Role Graphs at Procedure Entry

Our role analysis creates the set of role graphs at procedure entry point from the initial context $\mathsf{context}(\mathsf{proc})$. This is simple because role graphs and the initial context have similar abstraction relations (Sections 6.6.1 and 6.7.1). The difference is that

$$
\begin{aligned}
[\![\texttt{entry}\bullet]\!] = \Big\{ \langle H, \rho, K, \tau, E\rangle \;\Big|\; & \\
& P : \{\mathsf{proc}\} \times \{\mathsf{param}_i(\mathsf{proc})\}_i \to N, P \subseteq H_{\mathsf{IC}} \\
& H_0 = (H_{\mathsf{IC}} \setminus \{\mathsf{proc}\} \times \mathsf{param}(\mathsf{proc}) \times N) \cup P \\
& n_i = P(\mathsf{proc}, \mathsf{param}_i(\mathsf{proc})) \\
& H_1 \subseteq H_0 \\
& H_1 \setminus H_0 \subseteq \{\langle n', f, n''\rangle \mid \{n_1, n_2\} \cap \{n_i\}_i \neq \emptyset\} \\
& \forall j : \mathsf{localCheck}(n_j, \langle H, \rho, K\rangle, \mathsf{nodes}(H_1)) \\
& H_1 \overset{n_1}{\|} H_2 \overset{n_2}{\|} \cdots \overset{n_p}{\|} H \\
& \rho = \rho_{\mathsf{IC}} \\
& K = K_{\mathsf{IC}} \\
& \tau = \rho_{\mathsf{IC}} \\
& E = \emptyset \Big\}
\end{aligned}
$$

Figure 6-24: The Set of Role Graphs at Procedure Entry

parameters in role graphs point to exactly one node, and parameter nodes are onstage nodes in role graphs which means that all their edges are "must" edges.

Figure 6-24 shows the construction of the initial set of role graphs. First the graph H_0 is created such that every parameter $\mathsf{param}_i(\mathsf{proc})$ references exactly one parameter node n_i. Next graph H_1 is created by using $\mathsf{localCheck}$ to ensure that parameter nodes have the appropriate number of edges. Finally, the instantiation is performed on parameter nodes to ensure acyclicity constraints if the initial context does not make them explicit already.

Statement s	Transition	Constraints
`x = y.f`	$\langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_x\}, \rho, K, \tau, E\rangle \overset{\mathsf{st}}{\Longrightarrow} \langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_f\}, \rho, K, \tau, E\rangle$	$\langle \mathsf{proc}, \mathtt{y}, n_y\rangle, \langle n_y, f, n_f\rangle \in H$ $\tau(n_f) \in \mathsf{read}(\mathsf{proc})$
`x = y.f`	$\langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_x\}, \rho, K, \tau, E\rangle \overset{\mathsf{st}}{\Longrightarrow} \bot_G$	$\langle \mathsf{proc}, \mathtt{y}, n_y\rangle, \langle n_y, f, n_f\rangle \in H$ $\tau(n_f) \notin \mathsf{read}(\mathsf{proc})$
`x.f = y`	$\langle H \uplus \{n_x, f, n_f\}, \rho, K, \tau, E\rangle \overset{\mathsf{st}}{\Longrightarrow} \langle H \uplus \{n_x, f, n_y\}, \rho, K, \tau, E\rangle$	$\langle \mathsf{proc}, \mathtt{x}, n_x\rangle, \langle \mathsf{proc}, \mathtt{y}, n_y\rangle \in H$ $\langle \tau(n_x), f, \tau(n_y)\rangle \in \mathsf{mayWr}(\mathsf{proc})$
`x.f = y`	$\langle H \uplus \{n_x, f, n_f\}, \rho, K, \tau, E\rangle \overset{\mathsf{st}}{\Longrightarrow} \langle H \uplus \{n_x, f, n_y\}, \rho, K, \tau, E'\rangle$	$\langle \mathsf{proc}, \mathtt{x}, n_x\rangle, \langle \mathsf{proc}, \mathtt{y}, n_y\rangle \in H$ $\langle \tau(n_x), f, \tau(n_y)\rangle \in \cup_i \mathsf{mustWr}_i(\mathsf{proc})$ $E' = \mathsf{updateWr}(E, \langle \tau(n_x), f, \tau(n_y)\rangle)$
`x.f = y`	$\langle H \uplus \{n_x, f, n_f\}, \rho, K, \tau, E\rangle \overset{\mathsf{st}}{\Longrightarrow} \bot_G$	$\langle \mathsf{proc}, \mathtt{x}, n_x\rangle, \langle \mathsf{proc}, \mathtt{y}, n_y\rangle \in H$ $\langle \tau(n_x), f, \tau(n_y)\rangle \notin \mathsf{mayWr}(\mathsf{proc}) \cup$ $\cup_i \mathsf{mustWr}_i(\mathsf{proc})$
`x = new`	$\langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_x\}, \rho, K, \tau, E\rangle \overset{\mathsf{st}}{\Longrightarrow} \langle H \uplus \{\mathsf{proc}, \mathtt{x}, n_n\}, \rho, K, \tau', E\rangle$	n_n fresh $\tau' = \tau[n_n \mapsto \mathsf{NEW}]$

$$\mathsf{updateWr}(E, \langle n_1, f, n_2\rangle) = E[\langle n_1, f\rangle \mapsto n_2]$$

Figure 6-25: Verifying Load, Store, and New Statements

Verifying Basic Statements

To ensure that a procedure conforms to its transfer relation the analysis uses the initial context transformation τ to assign every Load and Store statement to a declared effect. Figure 6-25 shows new symbolic execution of Load, Store and New statements.

The symbolic execution of Load statement `x=y.f` makes sure that the node being loaded is recorded in some read effect. If this is not the case, an error is reported.

The symbolic execution of the Store statement `x.f=y` first retrieves nodes $\tau(n_x)$ and $\tau(n_y)$ in the initial role graph $\mathsf{context}$ that correspond to nodes n_x and n_y in the current role graph. If the effect $\langle\tau(n_x), f, \tau(n_y)\rangle$ is declared as a may write effect the execution proceeds as usual. Otherwise, the effect is used to update the list E of must-write effects. The list E is checked at the end of procedure execution.

The symbolic execution of the New statement updates the initial context transformation τ assigning $\tau(n_n) = \mathsf{NEW}$ for the new node n_n.

The τ transformation is similarly updated during other abstract heap operations. Instantiation of node n' into node n_0 assigns $\tau(n_0) = \tau(n')$, split copies values of τ into the new set of isomorphic nodes, and normalization does not merge nodes n_1 and n_2 if $\tau(n_1) \neq \tau(n_2)$.

Verifying Procedure Postconditions

At the end of the procedure, the analysis verifies that $\rho(n_i) = \mathsf{postR}_i(\mathsf{proc})$ where $\langle\mathsf{proc}, \mathsf{param}_i(\mathsf{proc}), n_i\rangle \in H$, and then performs node check on all onstage nodes using predicate $\mathsf{nodeCheck}(n, \langle H, \rho, K\rangle, \mathsf{nodes}(H))$ for all $n \in \mathsf{onstage}(H)$.

At the end of the procedure, the analysis also verifies that every performed effect in $E = \{e_1, \ldots, e_k\}$ can be attributed to exactly one declared must effect. This means that $k = \mathsf{mustWrNo}(\mathsf{proc})$ and there exists a permutation s of set $\{1, \ldots, k\}$ such that $e_{s(i)} \in \mathsf{mustWr}_i(\mathsf{proc})$ for all i.

6.7.3 Analyzing Call Sites

The set of role graphs at the procedure call site is updated based on the procedure transfer relation as follows. Consider procedure proc containing call site $p \in N_{\mathsf{CFG}}(\mathsf{proc})$ with procedure call $\mathsf{proc}'(x_1, \ldots, x_p)$. Let $\langle H_{\mathsf{IC}}, \rho_{\mathsf{IC}}, K_{\mathsf{IC}}\rangle = \mathsf{context}(\mathsf{proc}')$ be the initial context of the callee.

Figure 6-26 shows the transfer function for procedure call sites. It has the following phases:

1. **Parameter Check** ensures that roles of parameters conform to the roles expected by the callee proc'.
2. **Context Matching** ($\mathsf{matchContext}$) ensures that the caller's role graphs represent a subset of concrete heaps represented by $\mathsf{context}(\mathsf{proc}')$. This is done by deriving a mapping μ from the caller's role graph to $\mathsf{nodes}(H_{\mathsf{IC}})$.
3. **Effect Instantiation** ($\xrightarrow{\mathsf{FX}}$) uses effects $\mathsf{mayWr}(\mathsf{proc}')$ and $\mathsf{mustWr}_i(\mathsf{proc}')$ in order to approximate all structural changes to the role graph that proc' may

```
$[\![\mathsf{proc}'(x_1,\ldots,x_p)]\!](\mathcal{G}) =$
    if $\exists G \in \mathcal{G} : \neg\mathsf{paramCheck}(G)$ then $\{\bot_G\}$
      else try $\mathcal{G}_1 = \mathsf{matchContext}(\mathcal{G})$
          if failed then $\{\bot_G\}$
           else $\{G'' \mid \langle G,\mu\rangle \in \mathcal{G}_1$
                  $\langle \mathsf{addNEW}(G),\mu\rangle \xrightarrow{\mathsf{FX}} \langle G',\mu\rangle \xrightarrow{\mathsf{RR}} G''\}$

$\mathsf{paramCheck}(\langle H,\rho,K,\tau,E\rangle)$ iff
    $\forall n_i : \mathsf{nodeCheck}(n_i, G, \mathsf{offstage}(H) \cup \{n_i\}_i)$
    $n_i$ are such that $\langle \mathsf{proc}, x_i, n_i\rangle \in H$

$\mathsf{addNEW}(\langle H,\rho,K,\tau,E\rangle) =$
    $\langle H \cup \{n_0\} \times F \times \{\mathsf{null}\},$
    $\rho[n_0 \mapsto \mathsf{unknown}],$
    $K[n_0 \mapsto s],$
    $\tau[n_0 \mapsto \mathsf{NEW}],$
    $E\rangle$
where $n_0$ is fresh in $H$
```

Figure 6-26: Procedure Call

perform.

4. **Role Reconstruction** ($\xrightarrow{\mathsf{RR}}$) uses final roles for parameter nodes and global role declarations $\mathsf{postR}_i(\mathsf{proc}')$ to reconstruct roles of all nodes in the part of the role graph representing modified region of the heap.

The parameter check requires $\mathsf{nodeCheck}(n_i, G, \mathsf{offstage}(H) \cup \{n_i\}_i)$ for the parameter nodes n_i. The other three phases are explained in more detail below.

Context Matching

Figure 6-27 shows our context matching function. The matchContext function takes a set $\mathcal{G}$ of role graphs and produces a set of pairs $\langle G,\mu\rangle$ where $G = \langle H,\rho,K,\tau,E\rangle$ is a role graph and μ is a homomorphism from H to H_{IC}. The homomorphism μ guarantees that $\alpha^{-1}(G) \subseteq \alpha_0^{-1}(\mathsf{context}(\mathsf{proc}'))$ since the homomorphism h_0 from Definition 28 can be constructed from homomorphism h in Definition 22 by putting $h_0 = \mu \circ h$. This implies that it is legal to call proc' with any concrete graph represented by G.

The algorithm in Figure 6-27 starts with empty maps $\mu = \mathsf{nodes}(G) \times \{\bot\}$ and extends μ until it is defined on all $\mathsf{nodes}(G)$ or there is no way to extend it further. It proceeds by choosing a role graph $\langle H,\rho,K,\tau,E\rangle$ and node n_0 for which the mapping μ is not defined yet. It then finds candidates in the initial context that n_0 can be mapped to. The candidates are chosen to make sure that μ remains a homomorphism. The accessibility requirement—that a procedure may see no nodes with incorrect role—is enforced by making sure that nodes in inaccessible are never mapped into nodes

$$\mathsf{matchContext}(\mathcal{G}) = \mathsf{match}(\{\langle G, \mathsf{nodes}(G) \times \{\bot\}\rangle \mid G \in \mathcal{G}\})$$

$$\mathsf{match} : \mathcal{P}(\mathsf{RoleGraphs} \times (N \cup \{\bot\})^N) \rightharpoonup \mathcal{P}(\mathsf{RoleGraphs} \times N^N)$$

```
$\mathsf{match}(\Gamma) =$
 $\Gamma_0 := \{\langle G, \mu\rangle \in \Gamma \mid \mu^{-1}(\bot) \neq \emptyset\}$;
 if $\Gamma_0 = \emptyset$ then return $\Gamma$;
 $\langle\langle H, \rho, K, \tau, E\rangle, \mu\rangle :=$ choose $\Gamma_0$;
 $\Gamma' = \Gamma \setminus \langle\langle H, \rho, K, \tau, E\rangle, \mu\rangle$;
 $\mathsf{paramnodes} := \{n \mid \exists i : \langle \mathsf{proc}, x_i, n\rangle \in H\}$;
 $\mathsf{inaccessible} := \mathsf{onstage}(H) \setminus \mathsf{paramnodes}$;
 $n_0 :=$ choose $\mu^{-1}(\bot)$;
 $\mathsf{candidates} := \{n' \in \mathsf{nodes}(H_{\mathsf{IC}}) \mid$
           $(n_0 \notin \mathsf{inaccessible}$ and $\rho_{\mathsf{IC}}(n') = \rho(n_0))$ or
           $(n_0 \in \mathsf{inaccessible}$ and $n' \notin \mathsf{read}(\mathsf{proc}'))\}$
     $\bigcap_{\substack{\langle n_0, f, n\rangle \in H \\ \mu(n) \neq \bot}} \left\{n' \,\middle|\, \langle n', f, \mu(n)\rangle \in H_{\mathsf{IC}}\right\}$
     $\bigcap_{\substack{\langle n, f, n_0\rangle \in H \\ \mu(n) \neq \bot}} \left\{n' \,\middle|\, \langle \mu(n), f, n'\rangle \in H_{\mathsf{IC}}\right\}$;
 if $\mathsf{candidates} = \emptyset$ then fail ;
 if $\mathsf{candidates} = \{n'_0\}, K(n_0) = s, K_{\mathsf{IC}}(n'_0) = i, \mu^{-1}(n'_0) = \emptyset$
    then $\mathsf{match}(\Gamma' \cup \{\langle G', \mu[n_1 \mapsto n'_0]\rangle \mid \langle H, \rho, K, \tau, E\rangle \overset{n_1}{\underset{n_0}{\Uparrow}} G'\})$
  else $n'_0 :=$ choose $\{n' \in \mathsf{candidates} \mid K(n') = s$ or
              $(K(n_0) = i, \mu^{-1}(n') = \emptyset)\}$
      $\mathsf{match}(\Gamma' \cup \langle\langle H, \rho, K, \tau, E\rangle, \mu[n_0 \mapsto n'_0]\rangle)$;
```

Figure 6-27: The Context Matching Algorithm

in read for the callee. As long as this requirement holds, nodes in inaccessible can be mapped onto nodes of any role since their role need not be correct anyway. We generally require that the set $\mu^{-1}(n_0')$ for individual node n_0' in the initial context contain at most one node, and this node must be individual. In contrast, there might be many individual and summary nodes mapped onto a summary node. We relax this requirement by performing instantiation of a summary node of the caller if, at some point, that is the only way to extend the mapping μ (this corresponds to the first recursive call in the definition of match in Figure 6-27).

The algorithm is nondeterministic in the order in which nodes to be matched are selected. One possible ordering of nodes is depth-first order in the role graph starting from parameter nodes. If some nondeterministic branch does not succeed, the algorithm backtracks. The function fails if all branches fail. In that case the procedure call is considered illegal and $\perp_G$ is returned. The algorithm terminates since every procedure call lexicographically increases the sorted list of numbers $|\mu[\mathsf{nodes}(H)]|$ for $\langle\langle H, \rho, K, \tau, E\rangle, \mu\rangle \in \Gamma$.

Effect Instantiation

The result of the matching algorithm is a set of pairs $\langle G, \mu\rangle$ of role graphs and mappings. These pairs are used to instantiate procedure effects in each of the role graphs of the caller. Figure 6-28 gives rules for effect instantiation. The analysis first verifies that the region read by the callee is included in the region read by the caller. Then it uses map μ to find the inverse image S of the performed effects. The effects in S are grouped by the source n and field f. Each field $n.f$ is applied in sequence. There are three cases when applying an effect to $n.f$:

1. There is only one node target of the write in $\mathsf{nodes}(H)$ and the effect is a must write effect. In this case we do a strong update.
2. The condition in 1) is not satisfied, and the node n is offstage. In this case we conservatively add all relevant edges from S to H.
3. The condition in 1) is not satisfied, but the node n is onstage i.e. it is a parameter node[3]. In this case there is no unique target for $n.f$, and we cannot add multiple edges either as this would violate the invariant for onstage nodes. We therefore do case analysis choosing which effect was performed last. If there are no must effects that affect n, then we also consider the case where the original graph is unchanged.

Role Reconstruction

Procedure effects approximate structural changes to the heap, but do not provide information about role changes for non-parameter nodes. We use the role reconstruction algorithm $\xrightarrow{\mathsf{RR}}$ in Figure 6-29 to conservatively infer possible roles of nodes after the procedure call based on role changes for parameters and global role definitions.

[3]Non-parameter onstage nodes are never affected by effects, as guaranteed by the matching algorithm.

$$\langle\langle H, \rho, K, \tau, E\rangle, \mu\rangle \xrightarrow{\mathsf{FX}} \langle\bot_G, \mu\rangle \text{ where } \quad \tau[\mu^{-1}[\mathsf{read}(\mathsf{proc}')]] \not\subseteq \mathsf{read}(\mathsf{proc})$$

$$\langle\langle H, \rho, K, \tau, E\rangle, \mu\rangle \xrightarrow{\mathsf{FX}} G_t \text{ where } \quad \tau[\mu^{-1}[\mathsf{read}(\mathsf{proc}')]] \subseteq \mathsf{read}(\mathsf{proc})$$

$$\langle H, \rho, K, \tau, E\rangle \overset{n_1,f_1}{\vdash} G_1 \vdash \cdots \overset{n_t,f_t}{\vdash} G_t$$

$$S = \{\langle n, f, n'\rangle \in H \mid \langle \mu(n), f, \mu(n')\rangle \in \mathsf{mayWr}(\mathsf{proc}') \cup \cup_i \mathsf{mustWr}_i(\mathsf{proc}')\}$$

$$\{\langle n_1, f_1\rangle, \ldots, \langle n_t, f_t\rangle\} = \{\langle n, f\rangle \mid \langle n, f, n'\rangle \in S\}$$

Single Write Effect Instantiation:

$$\langle H_1, \rho_1, K_1, \tau_1, E_1\rangle \overset{n,f}{\vdash} G'$$

iff

case	condition	result
deterministic effect	$\{n_1 \mid \langle n, f, n_1\rangle \in S\} = \{n_0\}$ and $\exists i : \langle \mu(n), f, \mu(n_0)\rangle \in \mathsf{mustWr}_i(\mathsf{proc}')$	$G' = \langle H_2, \rho_1, K_1, \tau_1, E_2\rangle$ $H_2 = H_1 \setminus \{\langle n, f, n_1\rangle \mid \langle n, f, n_1\rangle \in H_1\}$ $\cup\{\langle n, f, n_0\rangle\}$ $E_2 = \mathsf{updateWr}(E_1, \langle \tau(n), f, \tau(n_0)\rangle)$
nondeterministic effect for non-parameters	$\lvert\{n_1 \mid \langle n, f, n_1\rangle \in S\}\rvert > 1$ or $\exists n_1 : \langle \mu(n), f, \mu(n_1)\rangle \in \mathsf{mayWr}(\mathsf{proc}')$ $n \in \mathsf{offstage}(H)$ $\{\langle \tau(n), f, \tau(n_1)\rangle \mid \langle n, f, n_1\rangle \in S\} \subseteq \mathsf{mayWr}(\mathsf{proc})$	$G' = \langle H_2, \rho_1, K_1, \tau_1, E_2\rangle$ $H_2 = \mathsf{orem}(H_1)\cup$ $\{\langle n, f, n_1\rangle \mid \langle n, f, n_1\rangle \in S\}$
	$\lvert\{n_1 \mid \langle n, f, n_1\rangle \in S\}\rvert > 1$ or $\exists n_1 : \langle \mu(n), f, \mu(n_1)\rangle \in \mathsf{mayWr}(\mathsf{proc}')$ $n \in \mathsf{offstage}(H)$ $\{\langle \tau(n), f, \tau(n_1)\rangle \mid \langle n, f, n_1\rangle \in S\} \not\subseteq \mathsf{mayWr}(\mathsf{proc})$	$G' = \bot_G$
nondeterministic effect for parameters	$\lvert\{n_1 \mid \langle n, f, n_1\rangle \in S\}\rvert > 1$ or $\exists n_1 : \langle \mu(n), f, \mu(n_1)\rangle \in \mathsf{mayWr}(\mathsf{proc}')$ $n \notin \mathsf{offstage}(H)$ $\{\langle \tau(n), f, \tau(n_1)\rangle \mid \langle n, f, n_1\rangle \in S\} \subseteq \mathsf{mayWr}(\mathsf{proc})$	$G' = \langle H_2, \rho_1, K_1, \tau_1, E_2\rangle$ $H_0 = H_1 \setminus \{\langle n, f, n_1\rangle \mid \langle n, f, n_1\rangle \in H_1\}$ $H_2 = H_1$ or $H_2 = H_0 \cup \{\langle n, f, n_1\rangle\}$ $\langle n, f, n_1\rangle \in S$
	$\neg(\{n_1 \mid \langle n, f, n_1\rangle \in S\} = \{n_1\}$ and $\exists i : \langle \mu(n), f, \mu(n_0)\rangle \in \mathsf{mustWr}_i(\mathsf{proc}'))$ $n \notin \mathsf{offstage}(H)$ $\{\langle \tau(n), f, \tau(n_1)\rangle \mid \langle n, f, n_1\rangle \in S\} \not\subseteq \mathsf{mayWr}(\mathsf{proc})$	$G' = \bot_G$

$$\mathsf{orem}(H_1) = \begin{cases} H_1 \setminus \{\langle n, f, n'\rangle \mid \langle n, f, n'\rangle \in H_1\}, \text{ if } \exists i\, \exists n' : \langle \mu(n), f, \mu(n')\rangle \in \mathsf{mustWr}_i(\mathsf{proc}') \\ H_1, \text{ otherwise} \end{cases}$$

Figure 6-28: Effect Instantiation

$$\langle\langle H,\rho,K,\tau,E\rangle,\mu\rangle \xrightarrow{\mathsf{RR}} \langle H',\rho',K',\tau',E'\rangle$$

$$\begin{array}{l}
\langle \mathsf{proc}, x_i, n_i\rangle \in H \\
N_0 = \mu^{-1}[\mathsf{read}(\mathsf{proc}')] \\
s : N_0 \times R \to N \text{ where } s(n,r) \text{ are all different nodes fresh in } H \\
\rho' = \rho \setminus (N_0 \times R) \cup \{\langle s(n,r), r\rangle \mid n \in N_0, r \in R\} \\
\qquad\qquad \setminus(\{n_i\}_i \times R) \cup \{\langle n_i, \mathsf{postR}_i(\mathsf{proc})\rangle\} \\
K'(s(n,r)) = K(n) \\
\tau'(s(n,r)) = \tau(n) \\
E' = E \\
H_0 = H \setminus \{\langle n_1, f, n_2\rangle \mid n_1 \in N_0 \text{ or } n_2 \in N_0\} \\
\qquad\qquad \cup \{\langle s(n_1,r_1), f, s(n_2,r_2)\rangle \mid \langle n_1,f,n_2\rangle \in H, \langle r_1,f,r_2\rangle \in \mathsf{RRD}\} \\
\qquad\qquad \cup \{\langle n_1, f, s(n_2,r_2)\rangle \mid \langle n_1,f,n_2\rangle \in H, \langle \rho_{\mathsf{IC}}(\mu(n_1)),f,r_2\rangle \in \mathsf{RRD}\} \\
\qquad\qquad \cup \{\langle s(n_1,r_1), f, n_2\rangle \mid \langle n_1,f,n_2\rangle \in H, \langle r_1,f,\rho_{\mathsf{IC}}(\mu(n_2))\rangle \in \mathsf{RRD}\} \\
H' = \mathsf{GC}(H_0)
\end{array}$$

Figure 6-29: Call Site Role Reconstruction

Role reconstruction first finds the set N_0 of all nodes that might be accessed by the callee since these nodes might have their roles changed. Then it splits each node $n \in N_0$ into $|R|$ different nodes $\rho(n,r)$, one for each role $r \in R$. The node $\rho(n,r)$ represents the subset of objects that were initially represented by n and have role r after procedure executes. The edges between nodes in the new graph are derived by simultaneously satisfying 1) structural constraints between nodes of the original graph; and 2) global role constraints from the role reference diagram. The nodes $\rho(n,r)$ not connected to the parameter nodes are garbage collected in the role graph. In practice, we generate nodes $\rho(n,r)$ and edges on demand starting from parameters making sure that they are reachable and satisfy both kinds of constraints.

6.8 Extensions

This section presents extensions of the basic role system. The *multislot* extension allows statically unbounded number of aliases for objects. *Root variables* allow stack frames to be treated as the source of aliases in role definitions. *Singleton roles* allow role declarations to specify that there is only one object of a given role. The extension for *cascading role changes* allows the analysis to verify more complex role changes. The extension to *partial roles* allows mutually independent role properties to be specified separately and then combined.

6.8.1 Multislots

A multislot $\langle r', f\rangle \in \mathsf{multislots}(r)$ in the definition of role r allows any number of aliases $\langle o', f, o\rangle \in H_c$ for $\rho_c(o') = r'$ and $\rho_c(o) = r$. We require multislots $\mathsf{multislots}(r)$ to be disjoint from all $\mathsf{slot}_i(r)$. To handle multislots in role analysis we relax the condition 5) in Definition 22 of the abstraction relation by allowing h to map more than one concrete edge $\langle o', f, o \rangle$ onto abstract edge $\langle n', f, n \rangle \in H$ terminating at an onstage node n provided that $\langle \rho(n'), f \rangle \in \mathsf{multislots}(\rho(n))$. The nodeCheck and expansion relation $\preceq$ are then extended appropriately. Note that a role graph does not represent the exact number of references that fill each multislot. The analysis therefore does not attempt to recognize actions that remove the last reference from the multislot. Once an object plays a role with a multislot, all subsequent roles that it plays must also have the multislot.

6.8.2 Root Variables

Root variables allow roles to be defined not only by heap references from other nodes but also by references from procedure variables. The root variables are treated like heap references for the purpose of role consistency; they are references from stack frame objects. A procedure with root variables induces a role with fields corresponding to root variables and no slots.

Example 32 Let us reconsider the scheduler example in Figure 6-2. We can require the `LiveHeader` node to be referenced by the root variable `processes` in the procedure `main`, and `RunningHeader` to be referenced by the root variable `running` in the following way.

```
role LiveHeader {
  fields first : LiveList | null;
  slots  main.processes;
}
role RunningHeader {
  fields next : RunningProc | RunningHeader,
         prev : RunningProc | RunningHeader;
  slots  main.running,
         RunningHeader.next | RunningProc.next,
         RunningHeader.prev | RunningProc.prev;
  identities next.prev, prev.next;
}

procedure main()
rootvar processes : LiveHeader | null,
        running   : RunningHeader | null;
{ ... }
```

This implicitly generates a role definition for the `main` procedure.

```
role main {
  fields processes : LiveHeader,
         running : RunningHeader;
}
```

△

```
role H { // header node
  fields next : H | N;
  slots  H.next | N.next;
}
role N { // internal node
  fields next : H | N;
  slots  H.next | N.next;
}
```

Figure 6-30: Roles for Circular List

6.8.3 Singleton Roles

Singleton roles are a simple way to improve the precision of role specifications and role analysis by indicating roles for which there is only a single heap object of that role. Singleton roles are often referred to from root variables.

We say that the predicate $\mathsf{singleton}(r)$ holds for role $r \in R$ if $|\rho_c^{-1}(r)| \leq 1$ for every valid concrete role assignment ρ_c of a heap created by the program. In essence, this predicate allows distinguishing between individual objects and sets of objects in role definitions.

Example 33 The intention of the definition in Figure 6-30 is to specify a circular singly linked list with a header node. However, the specification in Figure 6-30 is too general. For example, the graph in Figure 6-31 satisfies this specification. If we require $\mathsf{singleton}(\mathtt{H})$, then the graph in Figure 6-31 does not satisfy role declarations any more. $\triangle$

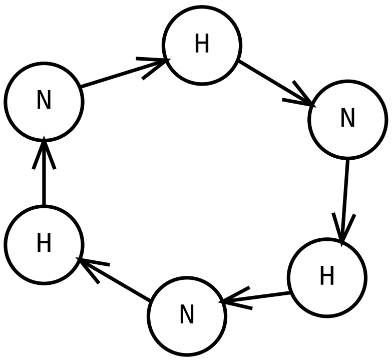


Figure 6-31: An Instance of Role Declarations

The developer can specify values of singleton predicate explicitly. In some cases the analysis alone can infer this information using the following rules:

- procedure activation records are singleton if they are not members of a cycle the call graph;
- if the roles $R_s \in R$ are singleton and $r' \in R$ is such that one of the following criteria holds:
  - there exists $f \in F$ such that $\mathsf{field}_f(r) \subseteq R_s$, or
  - there exists i such that $\mathsf{slot}_i(r') \subseteq R_s$,

  then r' is a singleton role as well.

When analyzing programs with singleton roles, the role analysis maintains the invariant that there is at most one node for each singleton role r by preventing multiple nodes with role r to go offstage. When traversing data structures, the singleton constraint eliminates cases in where two nodes with a singleton role are brought onstage.

A natural generalization of singleton roles arises in the context of *parametrized roles* [138]. The extension to parametrized roles is orthogonal to the other aspects of roles and we do not consider it in this chapter.

6.8.4 Cascading Role Changes

In some cases it is desirable to change roles of an entire set of offstage objects without bringing them onstage. We use the statement `setRoleCascade`$(x_1 : r_1, \ldots, x_n : r_n)$ to perform such *cascading role change* of a set of nodes. The need for cascading role changes arises when roles encode reachability properties.

Example 34 Procedure `main` in Figure 6-32 has two root variables, `buffer` and `work`, each being a root for a singly linked acyclic list. Elements of the first list have `BufferNode` role and elements of the second list have `WorkNode` role. At some point procedure swaps the root variables `buffer` and `work`, which requires all nodes in both lists to change the roles. These role changes are triggered by the `setRoleCascade` statement. The statement indicates new roles for onstage nodes, and the analysis cascades role changes to offstage nodes. $\triangle$

Given a role graph $\langle H, \rho, K, E \rangle$ cascading role change finds a new valid role assignment ρ' where the onstage nodes have desired roles and the roles of offstage nodes are adjusted appropriately. Figure 6-33 shows abstract execution of the `setRoleCascade` statement. Here $\mathsf{neighbors}(n, H)$ denotes nodes in H adjacent to n. The condition $\mathsf{cascadingOk}(n, H, \rho, K, \rho')$ makes sure it is legal to change the role of node n from $\rho(n)$ to $\rho'(n)$ given that the neighbors of n also change role according to ρ'. This check resembles the check for `setRole` statement in Section 6.6.2. Let $r = rho(n)$ and $r' = \rho'(n)$. Then $\mathsf{cascadingOk}(n, H, \rho, K, \rho')$ requires the following conditions:

1. $\langle n, f, n_1 \rangle \in H$ implies $\rho'(n_1) \in \mathsf{field}_f(r')$;

```
role BufferNode {
  fields next : BufferNode | null;
  slots  BufferNode.next | main.buffer;
  acyclic next;
}
role WorkNode {
  fields next : WorkNode | null;
         WorkNode.next | main.work;
  acyclic next;
}

procedure main()
rootvar buffer : BufferNode | null,
        work   : WorkNode | null;
auxvar x, y;
{
   // create buffer and work lists
    ...
   // swap buffer and work
   x = buffer;
   y = work;
   buffer = y;
   work = x;
   setRoleCascade(x:WorkNode, y:BufferNode);
}
```

Figure 6-32: Example of a Cascading Role Change

$\langle H, \rho, K, \tau, E\rangle \overset{\mathsf{st}}{\leadsto} \langle H, \rho', K, \tau, E\rangle$ $\mathsf{st} = \mathtt{setRoleCascade}(x_1 : r_1, \ldots, x_n : r_n)$	$n_i : \langle \mathsf{proc}, x_i, n_i\rangle \in H$ $\rho'(n_i) = r_i$ $\rho'(n) = \rho(n),\ n \in \mathsf{onstage}(H) \setminus \{n_i\}_i$ $N_0 = \{n \in \mathsf{offstage}(H) \mid \exists n' \in \mathsf{neighbors}(n, H) : \rho(n') \neq \rho'(n')\}$ $\forall n \in N_0 : \mathsf{cascadingOk}(n, H, \rho, K, \rho')$

Figure 6-33: Abstract Execution for `setRoleCascade`

2. $\mathsf{slotno}(r') = \mathsf{slotno}(r) = k$, and for every list $\langle n_1, f_1, n\rangle, \ldots, \langle n_k, f_k, n\rangle \in H$ if there is a permutation $p : \{1, \ldots, k\} \to \{1, \ldots, k\}$ such that $\langle \rho(n_i), f_i\rangle \in \mathsf{slot}_{p_i}(r)$, then there is a permutation $p' : \{1, \ldots, k\} \to \{1, \ldots, k\}$ such that $\langle \rho(n_i), f_i\rangle \in \mathsf{slot}_{p_i}(r')$;
3. identity relations were already satisfied or can be explicitly checked: $\langle f, g\rangle \in \mathsf{identities}(\rho'(n))$ implies
   (a) $\langle f, g\rangle \in \mathsf{identities}(\rho(n))$ or
   (b) for all $\langle n, f, n'\rangle \in H$: $K(n') = i$, and if $\langle n', g, n''\rangle \in H$ then $n'' = n$;
4. either $\mathsf{acyclic}(\rho'(n)) \subseteq \mathsf{acyclic}(\rho(n))$ or $\mathsf{acycCheck}(n, \langle H, \rho', K\rangle, \mathsf{offstage}(H))$.

In practice there may be zero or more solutions that satisfy constraints for a given cascading role change. Selecting any solution that satisfies the constraints is sound with respect to the original semantics. A useful heuristic for searching the solution space is to first explore branches with as few roles changed as possible. If no solutions are found, an error is reported.

6.8.5 Partial Roles

In this section we extend our framework to allow combining roles that specify mutually independent properties of objects. First we generalize field and slot constraints to allow specifying partial information about fields and slots of each role. We then give an alternative semantics of roles where each node is assigned a *set* of roles. A pleasant property of this semantics of roles is that the sets of roles applicable to each field can be defined as the greatest fixpoint of the recursive role definitions. We then sketch an extension of context matching and call site role reconstruction that allows procedures to be analyzed without specifying the full set of roles of objects in the initial role graphs.

Partial Roles and Role Sets

This section introduces *partial roles*. A partial role gives constraints only for a subset of fields and slots. We use the term *simple roles* to refer to non-partial roles considered so far.

```
role TR { // tree root
  fields left : TN | null,
         right : TN | null;
  left,right slots ;
}
role TN { // tree node
  fields left : TN | null,
         right : TN | null;
  left,right slots : TR.left | TR.right | TN.left | TN.right;
}
```

Figure 6-34: Definition of a Tree

Example 35 Consider the definition of a tree in Figure 6-34. This definition specifies that a data structure is a tree along the `left` and `right` fields, but does not constrain fields other than `left` and `right`. Similarly, the definition of a linked list in Figure 6-35 gives only requirements for the `next` field. Note how definition of `LH` specifies a

```
role LH { // list header
  fields next : NL | null;
  next slots ;
}
role LN { // list node
  fields next : LN | null;
  next slots LH.next | LN.next;
}
```

Figure 6-35: Definition of a List

partial "negative" slot constraint, namely the absence of a next field.

A definition for a threaded tree, for example, can leverage the preceding role definitions to define the composite data structure.

```
role LTN extends TN,LN { // linked tree node
  fields data : Stored;
}
```

Every object playing `LTN` role simultaneously plays `TN` and `LN` roles as well. In general, an object playing more roles satisfies more constraints. △

For partial roles, we change the convention that the fields not mentioned in a `fields` declaration are always constrained to be `null`. Instead, the absence of a field *f* implies no constraints on the roles that field *f* references. A slot constraint for a partial role r contains an additional set $\mathsf{scope}(r) = \{f_1, \ldots, f_k\}$ of fields that determine the scope of the slot constraints. A slot declaration gives complete aliases for references along $\mathsf{scope}(r)$ fields, but poses no requirements on aliases from other fields.

Partial role definitions can reuse previous role definitions using the `extends` keyword. We represent the `extends` relationships by the set of roles $\mathsf{subroles}(r)$ for each role r. A set $S \subseteq R$ is *closed* if $\mathsf{subroles}(r) \subseteq S$ for every $r \in S$.

6.8.6 Semantics of Partial Roles

To give the semantics of partial roles we define role-set assignment ρ_c^s to assign a closed set of roles to every object. We say that a role assignment ρ_c is a *choice* of a role-set assignment ρ_c^s iff $\rho_c(r) \in \rho_c^s(r)$ for every role $r \in R$. We first generalize $\mathsf{locallyConsistent}$ to take the role of the object o independently of role assignment ρ_c. This definition is identical to Definition 2 except that the role of the object o is r instead of $\rho_c(o)$.

Definition 36 $\mathsf{locallyConsistent}(o, H_c, \rho_c, r)$ *iff all of the following conditions are met.*

1) *For every field $f \in F$ and $\langle o, f, o' \rangle \in H_c$, $\rho_c(o') \in \mathsf{field}_f(r)$.*
2) *Let $\{\langle o_1, f_1 \rangle, \ldots, \langle o_k, f_k \rangle\} = \{\langle o', f \rangle \mid \langle o', f, o \rangle \in H_c\}$ be the set of all aliases of node o. Then $k = \mathsf{slotno}(r)$ and there exists some permutation p of the set $\{1, \ldots, k\}$ such that $\langle \rho_c(o_i), f_i \rangle \in \mathsf{slot}_{p_i}(r)$ for all i.*
3) *If $\langle o, f, o' \rangle \in H_c$, $\langle o', g, o'' \rangle \in H_c$, and $\langle f, g \rangle \in \mathsf{identities}(r)$, then $o = o''$.*
4) *It is not the case that graph H_c contains a cycle $o_1, f_1, \ldots, o_s, f_s, o_1$ where $o_1 = o$ and $f_1, \ldots, f_s \in \mathsf{acyclic}(r)$*

We now define the local role-set consistency as follows.

Definition 37 $\mathsf{locallyRSConsistent}(o, H_c, \rho_c^s)$ *iff for every $r \in \rho_c^s(o)$ there exists a choice ρ_c of ρ_c^s such that $\mathsf{locallyConsistent}(o, H_c, \rho_c, r)$. We say that a heap H_c is role-set consistent for a role-set assignment ρ_c^s if $\mathsf{locallyRSConsistent}(o, H_c, \rho_c^s)$ for every $o \in \mathsf{nodes}(H_c)$. We call such role-set assignment ρ_c^s a valid role-set assignment.*

We similarly extend the definitions of consistency for a given set of nodes from Definition 20.

The following observations follow from Definition 37:

1. if ρ_c^s is a valid role assignment, then $|\rho_c^s(o)| \geq 1$ for every object o, otherwise there would be no ρ_c which is a choice for ρ_c^s;
2. if $|\rho_c^s(o)| = 1$ for all $o \in \mathsf{nodes}(H_c)$, then heap consistency for partial roles is equivalent to heap consistency for simple roles.

Fixpoint Definition of the Greatest Role Assignment

We first show that the set of all valid role-set assignments has a least upper bound. We first define a partial order on functions from $\mathsf{nodes}(H_c)$ to $\mathcal{P}(R)$.

Definition 38 *$\rho^s_{c1} \sqsubseteq \rho^s_{c2}$ iff $\rho^s_{c1}(o) \subseteq \rho^s_{c2}(o)$ for every $o \in H_c$.*

We then introduce the pointwise union.

Definition 39

$$(\rho^s_{c1} \sqcup \rho^s_{c2})(o) = \rho^s_{c1}(o) \cup \rho^s_{c2}(o)$$

The union of two closed role-sets is a closed role-set, so the merge of two role-set assignments is still a role-set assignment. Moreover, if both role-set assignments are valid, the pointwise union is also a valid role-set assignment, as the following property shows.

Property 40 *Let ρ^s_{c1} and ρ^s_{c2} be valid role-set assignments for the heap H_c. Then $\rho^s_{c1} \sqcup \rho^s_{c2}$ is also a valid role assignment.*

The property holds because every role assignment ρ_c which is a choice of ρ^s_{c1} or a choice of ρ^s_{c2} is also a choice of $\rho^s_{c1} \sqcup \rho^s_{c2}$.

Because there is a finite number of role-set assignments, Property 40 implies the existence of the *greatest role-set assignment ρ^{sM}_c* which is the merge of all valid role assignments.

Definition 41 *Let $\rho^s_{c1}, \ldots, \rho^s_{cN}$ be all valid role assignments for the heap H_c. We define the greatest role assignment ρ^{sM}_c as*

$$\rho^{sM}_c = \rho^s_{c1} \sqcup \cdots \sqcup \rho^s_{cN}$$

Definition 42 *Let $\rho^s_c : \mathsf{nodes}(H_c) \to \mathcal{P}(R)$. Then $F(\rho^s_c) : \mathsf{nodes}(H_c) \to \mathcal{P}(R)$ is a defined by*

$$\begin{array}{rcl} F(\rho^s_c)(o) & = & \{r \in \rho^s_c(o) \mid \mathsf{subroles}(r) \subseteq \rho^s_c(o) \textit{ and} \\ & & \textit{there exists a choice } \rho_c \textit{ of } \rho^s_c \textit{ such that} \\ & & \mathsf{locallyConsistent}(o, H_c, \rho_c, r)\} \end{array}$$

Property 43 *The greatest role-set assignment for a concrete heap H_c is a greatest fixpoint of function F.*

Proof. It is easy to see that $F(\rho^s_{c1}) \sqsubseteq F(\rho^s_{c2})$ whenever $\rho^s_{c1} \sqsubseteq \rho^s_{c2}$. Also, $F(\rho^s_c) \sqsubseteq \rho^s_c$ and the empty role-set assignment $\rho^s_c(o) = \emptyset$ is a fixpoint of F.

Let ρ^s_{c0} be such that $\rho^s_{c0}(o) = R$ for all $o \in H_c$. Consider the sequence $F^i(\rho^s_{c0})$ for $i \geq 0$. There exists i_0 such that $F^i(\rho^s_{c0}) = \rho^s_{c*}$ for $i \geq i_0$ where ρ^s_{c*} is a fixpoint of F. Because $F(\rho^s_{c*})(o) = \rho^s_{c*}(o)$ for each o, it follows that ρ^s_{c*} is a valid role-set assignment. Moreover, if ρ^s_c is any other valid role-set assignment, then $\rho^s_c \sqsubseteq F^i(\rho^s_{c0})$ for every i, so $\rho^s_c \sqsubseteq \rho^s_{c*}$. We conclude that the fixpoint ρ^s_{c*} is the greatest valid role assignment ρ^{sM}_c. ∎

Expressibility of Partial Roles

The partial roles allow data structures to be described compositionally. Another nice property of partial roles is that there is a canonical role-set assignment ρ_c^{sM}. A drawback of considering only the greatest role-set assignment is that some data structure constraints are not expressible.

Example 44 The set of cycles of even length can be described using the following simple role definitions.

```
role Even {
  fields next : Odd;
  slots  Odd.next;
}
role Odd {
  fields next : Even;
  slots Even.next;
}
```

No odd length cycle satisfies this role assignment. Each even length cycle $o_1, \ldots, o_{2k}$ has two role assignments ρ_{c1} and ρ_{c2}, where $\rho_{c1}(o_{2i+1}) = \texttt{Odd}$ and $\rho_{c1}(o_{2i}) = \texttt{Even}$, whereas $\rho_{c2}(o_{2i+1}) = \texttt{Even}$ and $\rho_{c2}(o_{2i}) = \texttt{Odd}$.

On the other hand, the same role definitions have unique greatest role assignment $\rho_c^s = \rho_{c1}^s \sqcup \rho_{c2}^s$, where $\rho_c^s(o) = \{\texttt{Even}, \texttt{Odd}\}$ for all o. This role assignment is valid not only for even length cycles, but also for odd length cycles. △

The constraints that can be specified by partial roles and role-set assignments are similar to constraints that can be specified using simple roles and role assignments. In the absence of acyclicity constraints, given a set of partial role definitions, it is possible to exhibit a set of simple role definitions which capture the same constraints.

This construction introduces a simple role each closed set of partial roles, similar to the construction showing the equivalence of deterministic and nondeterministic finite state automata [146] or deterministic and nondeterministic finite tree automata [101, 65]. Construction is complicated by the form of our slot constraints, but can be done by introducing additional roles that simulate slot constraint conjunction. (The ability to perform conjunction of slot constraints is an easy consequence of the equivalence of slot constraints with the generalized slot constraints in Section 6.9.1.) The construction could also be performed for acyclicity constraints if we generalized them to specify a family of sets of fields and forbid cycles along paths with fields from each of the sets in the family.

Even after performing this construction, it remains the fact that partial roles induce additional partial order structure, which is not available in simple roles.

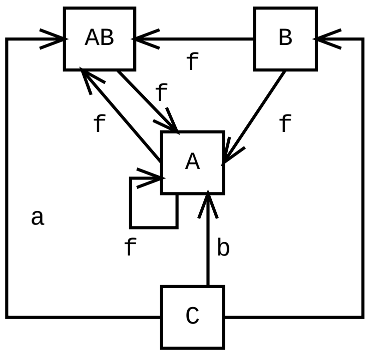
6.8.7 Role Subtyping

We now consider the problem of role subtyping at procedure call sites. A larger set of nodes for a node implies stronger constraints for that node. We would then expect a procedure call to be legal when the caller's role-sets are supersets of role-sets of the initial context. The problem is that a larger set $\rho^s_c(n)$, while implying a stronger constraint on the node n, implies *weaker* constraint on the nodes adjacent to n. The following example shows that the superset conditions on role-sets is in general not sufficient.

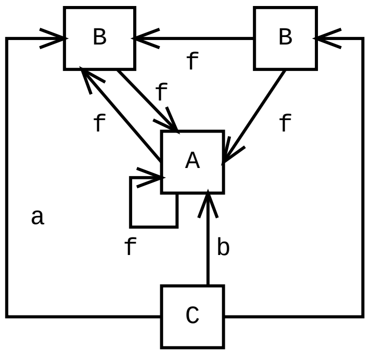
Example 45 Define roles `A` and `B` as follows:

```
role A {
  f slots A.f,
          B.f | A.f;
}
role B { }
role C { }
```

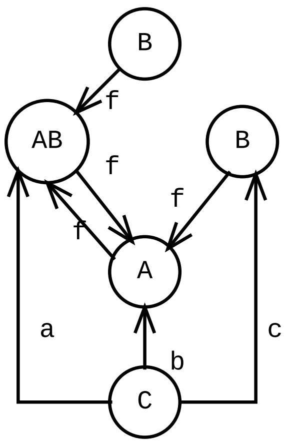
Consider the following role graph in the caller



and assume that the callee has the following initial role graph.



Clearly there is a homomorphism μ from the caller's role graph to the initial role graph such that $\rho^s_1(n) \supseteq \rho^s_2(\mu(n))$ for all nodes n. The following heap is an instance of the caller's role graph.



However, it is not possible to assign sets of roles to objects to make it an instance of the role graph in the initial context. △

The following property shows that a simple restriction on slot constraints makes the role-set inclusion criterion valid.

Property 46 *Let $\langle H, \rho^s, K\rangle$ and $\langle H_{\mathsf{IC}}, \rho^s_{\mathsf{IC}}, K_{\mathsf{IC}}\rangle$ be role graphs and $\mu : \mathsf{nodes}(H) \to \mathsf{nodes}(H_{\mathsf{IC}})$ a graph homomorphism such that:*

1. *$\rho^s(n) \supseteq \rho^s_{\mathsf{IC}}(\mu(n))$ for all $n \in \mathsf{nodes}(H)$;*
2. *if $\langle n_1, f, n_0\rangle \in H$, $r_0 \in \rho^s_{\mathsf{IC}}(\mu(n_0))$, $r_1 \in \rho^s(n_1)$, and $\langle r_1, f\rangle \in \mathsf{slot}_i(r_0)$ for some i, then $\langle r_2, f\rangle \in \mathsf{slot}_i(r_0)$ for some $r_2 \in \rho^s_{\mathsf{IC}}(\mu(n_1))$.*

Let H_c be a concrete heap such and ρ^s_{c1} a valid role-set assignment for H_c. Assume that h is a homomorphism from H_c to H such that $\rho^s_{c1}(o) = \rho^s(h(o))$ for all $o \in \mathsf{nodes}(H_c)$. Define

$$\rho^s_{c2}(o) = \rho^s_{\mathsf{IC}}(\mu(h(o)))$$

for all $o \in \mathsf{nodes}(H_c)$. Then ρ^s_{c2} is also a valid role-set assignment for H_c.

Proof. To show that ρ^s_{c2} is a valid role-set assignment for H_c, consider any object $o \in \mathsf{nodes}(H_c)$ and one of its roles $r_0 \in \rho^s_{c2}(o)$. Because $r_0 \in \rho^s_{c2}(o)$, identities and acyclicity constraints hold for o. We show that field and slot constraints hold as well.

To show that field constraints of r_0 hold, consider any edge $\langle o, f, o_1\rangle \in H_c$. Then $\langle n, f, n_1\rangle \in H_{\mathsf{IC}}$ where $n = \mu(h(o))$ and $n_1 = \mu(h(o_1))$. Because H_{IC} is a subgraph of the static role diagram, $\mathsf{field}_f(r_0) \cap \rho^s_{\mathsf{IC}}(n_1) \neq \emptyset$, otherwise the edge $\langle n, f, n_1\rangle$ would be superfluous. Since $\rho^s_2(o_1) = \rho^s_{\mathsf{IC}}(n_1)$ by definition of ρ^s_2, we have $\mathsf{field}_f(r_0) \cap \rho^s_2(o_1) \neq \emptyset$ which means that the field constraint for f is satisfied in H_c.

To show that slot constraints of r_0 hold, consider any edge $\langle o_1, f, o\rangle \in H_c$. Because ρ^s_{c1} is a valid role assignment and $r_0 \in \rho^s_{c1}(o)$, there exists slot i and role $r_1 \in \rho^s_{c1}(o_1)$ such that $\langle r_1, f\rangle \in \mathsf{slot}_i(r_0)$. By the assumption 2), since $\langle h(o_1), f, h(o)\rangle \in H$, $r_0 \in \rho^s_{\mathsf{IC}}(h(o))$ and $r_1 \in \rho^s(h(o_1))$, there exists $r_2 \in \rho^s_{\mathsf{IC}}(\mu(h(o_1))$ such that $\langle r_2, f\rangle \in \mathsf{slot}_i(r_0)$. Since $\rho^s_{\mathsf{IC}}(\mu(h(o_1)) = \rho^s_{c2}(o_1)$, it follows that the slot constraint of o is satisfied. ∎

The condition 2) in Property 46 can be replaced by a stronger but simpler condition.

Definition 47 *We say that role r_0* depends *on r_1 iff for some slot i, $\langle r_1, f\rangle \in \mathsf{slot}_i(r_0)$ and there exists another slot $j \neq i$ of role r_0 such that $\langle r_2, f\rangle \in \mathsf{slot}_j(r_0)$ for some role r_2.*

Property 48 *Let $\langle H, \rho^s, K\rangle$ and $\langle H_{\mathsf{IC}}, \rho^s_{\mathsf{IC}}, K_{\mathsf{IC}}\rangle$ be role graphs and $\mu : \mathsf{nodes}(H) \to \mathsf{nodes}(H_{\mathsf{IC}})$ a graph homomorphism such that:*

1') $\rho^s(n) \supseteq \rho^s_{\mathsf{IC}}(\mu(n))$ for all $n \in \mathsf{nodes}(H)$;

2') if $r_1 \in \rho^s(n) \setminus \rho^s_{\mathsf{IC}}(\mu(n))$ for some n, and r_0 depends on r_1, then for all $n' \in \mathsf{nodes}(H_{\mathsf{IC}})$, $r_0 \notin \rho^s_{\mathsf{IC}}(n')$.

Then the condition 2) of Property 46 is satisfied.

Proof. Let $\langle n_1, f, n\rangle \in H$, $r_0 \in \rho^s_{\mathsf{IC}}(n)$, and $r_1 \in \rho^s(H)$ and $\langle r_1, f\rangle \in \mathsf{slot}_i(r_0)$. If $r_1 \in \rho^s_{\mathsf{IC}}(\mu(n))$ then we can take $r_2 = r_1$ and the condition 2) is satisfied. Now assume $r_1 \in \rho^s(n) \setminus \rho^s_{\mathsf{IC}}(\mu(n))$. Since $r_0 \in \rho^s_{\mathsf{IC}}(n)$, by assumption 2'), r_0 does not depend on r_1. This means that i is the only slot of r_0 that contains the field f. Because the edge $\langle \mu(n_1), f, \mu(n)\rangle$ is in H_{IC}, and H_{IC}, it follows that $\langle r_2, f\rangle \in \mathsf{slot}_i(r_0)$ for some $r_2 \in \rho^s_{\mathsf{IC}}(n_1)$. This means that the condition 2) is satisfied. ∎

Based on previous properties we can derive a context matching algorithm that allows role graphs in the call site to have larger sets of roles than nodes in the initial context.

In order to further increase the precision of call site verification, we would like to preserve the larger larger set of role graphs in the caller. This is possible because procedure effects specify which object fields can be modified during execution of the caller. The role reconstruction algorithm for partial roles is similar to algorithm in Figure 6-29 except that it operates on sets of roles instead of individual roles. To consider how to preserve the wider set of roles, consider a role $r \in \rho^s(n) \setminus \rho^s_{\mathsf{IC}}(\mu(n))$. The role reconstruction splits n into a set of nodes each of which has assigned some role-set S. In the absence of write effects the algorithm would need to generate nodes with role-sets S that do not contain r. If the write effects imply that the role r cannot be violated, then only role-sets S containing r need to be generated, which increases the precision and reduces the size of role graphs after the procedure call. To compute the set of roles that are preserved, role reconstruction starts with sets $p(n) = \rho^s(n) \setminus \rho^s_{\mathsf{IC}}(\mu(n))$ assigned to each node n, and iteratively decreases sets $p(n)$ if a $r \in p(n)$ depends on a modified field or previously eliminated role.

We note that, similarly to multislots, partial roles allow a statically unbounded number of aliases. Whereas multislots explicitly give permission for existence of certain aliases, partial roles allow all the existence of aliases not mentioned in the role definition.

6.9 Decidability Properties of Roles

This section presents some further results about properties of roles. The first section proves decidability of the satisfiability problem for roles with only field and slot constraints. The second section proves undecidability of the implication problem for roles.

6.9.1 Roles with Field and Slot Constraints

In this section we closely examine more closely properties of roles defined using solely field and slot constraints. We ignore identity and acyclicity constraints in this and the following section.

We show that we can use more general form of slot constraints without changing the expressive power of roles. We then show how the generalized slot constraints can entirely replace the field constraints, which means that these constraints are not strictly necessary once the full set of role definitions is given. Finally we show decidability of the satisfaction problem for a set of roles containing only slot constraints.

Forms of Slot Constraints

The particular form of our slot constraints introduced in Section 6.4.1 may seem somewhat arbitrary. In this section we introduce a more general form of slot constraints and show that it can be reduced to our original role constraints. This observation gives insight into the nature of slot constraints and is used in further sections.

Definition 49 *A* generalized slot constraint *for role r, denoted* $\mathsf{gslot}(r)$*, is a list $c_1, \ldots, c_n$ of* incoming configurations*. Each incoming configuration c_s is a list of pairs $\langle r_{s1}, f_{s1}\rangle, \ldots, \langle r_{sq_s}, f_{sq_s}\rangle \in R \times F$ where q_s is the length of c_s.*

By abuse of notation, we write $\langle r_j, f_j\rangle \in c_s$ if $\langle r_j, f_j\rangle$ is a member of the list c_s where c_s represents the incoming configuration.

In addition to the role assignment $\rho_c : \mathsf{nodes}(H_c) \to R$, we introduce an incoming configuration assignment $\nu : \mathsf{nodes}(H_c) \to \mathcal{N}$. For each node o, the incoming configuration assignment selects an incoming configuration $c_{\nu(o)}$ of the the role $\rho_c(o)$. The local consistency is then defined as follows.

Definition 50 $\mathsf{locallyConsistent}(o, H_c, \rho_c, \nu)$ *holds for generalized roles iff the following conditions are met. Let $r = \rho_c(o)$.*

1) *For every field $f \in F$ and $\langle o, f, o'\rangle \in H_c$, $\rho_c(o') \in \mathsf{field}_f(r)$.*
2) *Let $\{\langle o_1, f_1\rangle, \ldots, \langle o_k, f_k\rangle\} = \{\langle o', f\rangle \mid \langle o', f, o\rangle \in H_c\}$ be the set of all aliases of node o and $s = \nu(o)$. Then $k = q_s$ and there exists a permutation p of the set $\{1, \ldots, k\}$ such that $\langle \rho_c(o_{p_i}), f_{p_i}\rangle = \langle r_{si}, f_{si}\rangle$ for $1 \leq i \leq k$ where $\langle r_{si}, f_{si}\rangle$ is the i-the element of the list in incoming configuration c_s.*

We say that the pair $\langle \rho_c, \nu\rangle$ of role assignment and incoming configuration assignment is valid for H_c iff $\mathsf{locallyConsistent}$ predicate holds for all nodes $o \in \mathsf{nodes}(H_c)$; the heap H_c is consistent if there exists a valid pair $\langle \rho_c, \nu\rangle$. A nonempty heap consistent with a given set of role definition is called a *model* for the role definitions.

Equivalence of Original and Generalized Slots

Our original slot constraints $\mathsf{slot}_i(r)$ for $1 \leq i \leq k$ where $k = \mathsf{slotno}(r)$ can be represented as generalized slot constraints with a list of all incoming configurations $c = \langle r_1, f_1 \rangle, \ldots, \langle r_k, f_k \rangle$ for $\langle r_i, f_i \rangle \in \mathsf{slot}_i(r)$, $1 \leq i \leq k$. This representation is a direct consequence of Definitions 50 and 2.

Conversely, given a set of role definitions with generalized slots, we can construct a set of role definitions with original slots as follows. Introduce a role r/c for each incoming configuration c of role r with generalized slot constraint. Let $\mathsf{origRoles}(r)$ denote the set of new roles r/c for all incoming configurations c of r. Define field and slot constraints for r/c as follows:

$$\mathsf{field}_f(r/c) = \bigcup \{\mathsf{origRoles}(r') \mid r' \in \mathsf{field}_f(r)\}$$

$$\mathsf{slot}_i(r/c) = \{\langle r_i/c', f_i \rangle \mid c' \text{ is an incoming configuration of } r_i\}$$

where $c = \langle r_1, f_1 \rangle, \ldots, \langle r_k, f_k \rangle$. Let role assignment ρ_c assign roles with generalized slots to objects and ν be the incoming configuration assignment such that $\mathsf{locallyConsistent}$ predicate holds for all heap objects. Define the assignment of original roles by

$$\rho'_c(o) = \rho_c(o)/\nu(o)$$

Then $\mathsf{locallyConsistent}$ predicate holds for the ρ'_c assigning original roles to objects.

We will use the generalized role constraints to establish the decidability of the satisfiability problem. We first show how to eliminate field constraints.

Eliminating Field Constraints

In this section we argue that the field constraints are mostly subsumed by slot constraints if the entire set of role definitions is given. The constraint $r' \notin \mathsf{field}_f(r)$ can be specified as $\langle r, f \rangle \notin \mathsf{slot}_i(r')$ for all slots i in the original slot constraints. In the generalized slot constraints this conditions is specified by making sure that $\langle r, f \rangle$ is not a member of any of the incoming configurations c of role r'. In order to allow this construction to work for `null` references, we introduce multislot declaration for null_R role by defining $\langle r, f \rangle \in \mathsf{multislots}(\mathsf{null}_R)$ iff $\mathsf{null}_R \in \mathsf{field}_f(r)$.

After this transformation, the field declarations will be satisfied whenever (generalized) slot constraints and null_R multislot constraint are satisfied. In the sequel we therefore ignore the field constraints.

Decidability of the Satisfiability Problem

In this section we show that is is decidable to determine if a given set of role definitions (containing only field and slot constraints) has a model. We show how to reduce this question to the solvability of an integer linear programming problem.

Assume a set of role definitions for roles $R = \{r_1, \ldots, r_n\}$. Let H_c be a concrete heap, ρ_c a role assignment and ν an incoming configuration assignment. Define the following nonnegative integer variables. For every i, where $1 \leq i \leq n$, let x_i be the number of nodes with role r_i:

$$x_i = |\{o \in \mathsf{nodes}(H_c) \mid \rho(o) = r_i\}|$$

Let y_{js} be the number of nodes with role $\rho_c(r_j)$ for which ν selects the incoming configuration c_s:

$$y_{js} = |\{o \in \mathsf{nodes}(H_c) \mid \rho(o) = r_j, \nu(o) = c_s\}|$$

We also introduce the values n_{fi} denoting the number of `null` references from objects with role r_i along the field f:

$$n_{fi} = |\{\langle o, f, \mathtt{null}\rangle \in H_c \mid \rho_c(o) = r_i\}|$$

Assume that $\mathsf{locallyConsistent}$ predicate holds for all objects $o \in \mathsf{nodes}(H_c)$. By partitioning the set of objects first by roles and then by incoming configurations of each role, we conclude that the following equations hold for $1 \leq j \leq n$:

$$\sum_{s=1}^{q_j} y_{js} = x_j \tag{6.1}$$

Next, let us count for each role r_i and each field $f \in F$, the number of f-references from objects in $\rho_c^{-1}(r_i)$. We assumed that each object has the field f, so counting the source of these references yields x_i. Out of these, n_{fi} are null references, and the remaining ones fill the slots of objects with incoming configurations that contain $\langle r_i, f\rangle$. We conclude that for each $f \in F$ and $1 \leq i \leq n$ the following linear equation holds:

$$x_i = n_{fi} + \sum_{\langle r_i, f\rangle \in c_s} y_{js} \tag{6.2}$$

Finally, for all $\langle r_i, f\rangle \notin \mathsf{multislots}(\mathsf{null}_R)$, we have

$$n_{fi} = 0 \tag{6.3}$$

We call equations 6.1, 6.2, and 6.3 the *characteristic equations* of role constraints.

We concluded that characteristic equations hold for each valid role and incoming configuration assignment. We now argue that a nontrivial solution of these equations implies the existence of a heap H_c, the role assignment ρ_c and incoming configuration assignment ν such that $\mathsf{locallyConsistent}$ predicate is satisfied for all objects of the heap.

Assume that there is a nontrivial solution of the characteristic equations. Construct a heap H_c with N nodes where $N = \sum_{i=1} x_i$. Partition the nodes of the heap into n classes and assign $\rho_c(o) = r_i$ for nodes in class i, such that the definition of x_i is satisfied for every i. This is possible by the choice of N. Next, partition each class $\rho_c^{-1}(r_i)$ into disjoint sets, one set for each incoming configuration, and assign $\nu(o) = c_s$ such that the definitions of y_{js} are satisfied. This is always possible because equation 6.1 holds. Next, add edges to graph H_c so that slot constraints are satisfied.

This can be done by a simple greedy algorithm which adds one edge at a time so that it does not violate any slot constraints. This construction is guaranteed to succeed because of equation 6.2. The condition 6.3 guarantees that the resulting graph null references will be present only for the fields for which they are allowed. The result is a heap H_c consistent with the role definitions.

The next theorem follows directly from the previous argument and the decidability of the integer linear programming problem.

Theorem 2 *It is decidable to determine if there exists a model for a given set of role definitions.*

In addition to showing the decidability, the preceding argument also illustrates that slot and field constraints are insensitive to graph operations that switch the source of a reference from object o_1 to object o_2, as long as $\rho_c(o_1) = \rho_c(o_2)$. This implies that certain heap properties are not expressible using slot and field constraints alone. In particular, slot constraints do not prevent cycles, which justifies introducing the acyclicity constraints into the role framework.

6.9.2 Undecidability of Model Inclusion

In this section we explore the decidability of the question "is the set of models of one set of role definitions S_1 included in the set of models of another set of role definitions S_2". This appears to be a more difficult problem than satisfiability of role definitions. Indeed, we proved in Section 6.9.1 that the satisfiability is decidable for a restricted class of role definitions; in this section we prove that the model inclusion problem is undecidable for acyclic models.

Our role specifications are interpreted with respect to graphs which need not be trees and can even contain cycles. It can therefore be expected that strong enough properties are undecidable for such broad class of models. A common technique to prove undecidability for problems on general graphs is to consider the class of graphs called *grids*.

We define a grid as a labelled graph with edges x along the x-axis and edges y along the y axis.

Definition 51 *A grid $m \times n$ where $m, n \geq 5$ is any graph isomorphic to the graph with nodes*

$$V = \{1, \ldots, m\} \times \{1, \ldots, n\}$$

and edges $E = E_r \cup E_d$ where

$$E_x = \{\langle\langle i, j\rangle, x, \langle i + j, j\rangle\rangle \mid 1 \leq i \leq m - 1, 1 \leq j \leq n\}$$

$$E_y = \{\langle\langle i, j\rangle, y, \langle i, j + 1\rangle\rangle \mid 1 \leq i \leq m, 1 \leq j \leq n - 1\}$$

The idea is to reduce the existence of a Turing machine computation history [182, 158] to the problem on graphs considered. The rules for computation history are local and thus can be expressed using slots and fields. However, it is not possible to use roles to directly express the condition that a graph is a grid. The problem is that the commutativity condition $o.x.y = o.y.x$ for grids cannot be captured using our role constraints, as the following reasoning shows.

Assume that there are role definitions which describe the class of grids. Since grids do not have any identities $\langle f, g\rangle$, we may assume that these role definitions do not contain identity declarations. Because the number of roles and incoming configurations is finite, there exists a sufficiently large grid E, a valid role assignment ρ_c and a valid incoming configuration assignment ν such that for some i, j where $2 < i < j$, all of the following conditions hold:

$$\begin{aligned}
\rho_c(\langle i, 2\rangle) &= \rho_c(\langle j, 2\rangle) \\
\rho_c(\langle i, 3\rangle) &= \rho_c(\langle j, 3\rangle) \\
\nu(\langle i, 2\rangle) &= \nu(\langle j, 2\rangle) \\
\nu(\langle i, 2\rangle) &= \nu(\langle j, 2\rangle)
\end{aligned}$$

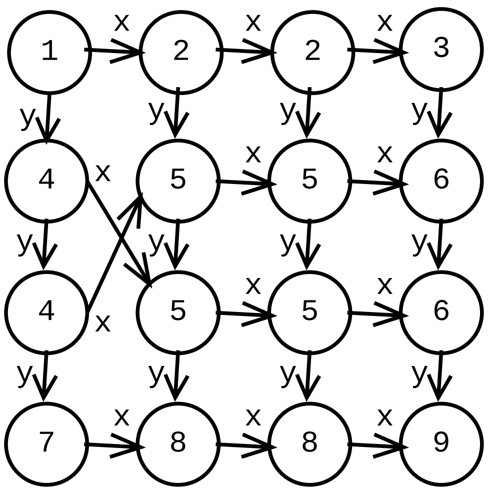


Figure 6-36: A Grid after Role Preserving Modification

Define a new graph E' in the following way (see Figure 6-36).

$$\begin{aligned}
E' &= (E \setminus \{\langle\langle i, 2\rangle, x, \langle i, 3\rangle\rangle, \langle\langle j, 2\rangle, x, \langle j, 3\rangle\rangle\}) \\
&\quad \cup \{\langle\langle i, 2\rangle, x, \langle j, 3\rangle\rangle, \langle\langle j, 2\rangle, x, \langle i, 3\rangle\rangle\}
\end{aligned}$$

We claim that the new graph E' also satisfies the same role and incoming configuration assignment. To see this, observe that the field and slot constraints remain satisfied because the new edges connect nodes with same roles as in E, there are no identities in role definitions, and the graph remains acyclic so acyclicity conditions cannot be violated. But E' is not isomorphic to a grid, because every isomorphism would have to be identity function on node $\langle 1,1\rangle$, and therefore also identity on all nodes $\langle 1,i\rangle$ for $i > 1$. Next, since y-edges in E' are the same as in E, the isomorphism would have to be identity function on all nodes, and this is not possible due to the change performed in the set of x-edges. We conclude there is no set of role definitions that captures the class of grids.

The idea of our undecidability construction is to use one set of role definitions S_1 to approximate the grid up to the commutativity condition $o.x.y = o.y.x$ as well as to encode the transitions of a Turing machine. We then use the another set of role definitions S_2 to express the *negation* of the commutativity condition. The models of S_1 are not included in models of S_2 if and only if there exists a model for S_1 which is not a model of S_2. Any such model will have to be a grid because it satisfies S_1 but not S_2, and the roles of S_1 will encode the accepting Turing machine computation history. Hence the question whether such a model exists will be equivalent to the existence of an accepting Turing machine computation history and the undecidability of model inclusion will follow from the undecidability of the halting problem.

Let us first consider how S_1 and S_2 define the grid used to encode the computation histories. Without the loss of generality, we restrict ourselves to models that are connected graphs. We define S_1 to be a refinement of the definition for a sparse matrix from Example 3, Figure 6-4. From properties in Section 6.4.3 we conclude that the connected models of E are graphs for which there exist $m, n \geq 3$ such that:

1. there is exactly one node `A1`, one node `A3`, one node `A7` and one node `A9`;
2. there are $m-2$ nodes `A2` (by the choice of m);
3. there are $m-2$ nodes `A8` because the acyclic lists along y establish bijection with `A2` nodes;
4. there are $n-2$ nodes `A4` (by the choice of n);
5. there are $n-2$ nodes `A6` because the acyclic lists along x establish bijection with `A4` nodes;
6. there are at least $\max(m-2, n-2)$ nodes `A5` (but not necessarily more than that).

The idea of role definitions S_2 is that if a graph satisfying S_1 is not a grid, then there must exist a node o such that $o.x.y \neq o.y.x$, which means that $o.x.y$ and $o.y.x$ can be assigned distinct roles. We construct S_2 to require the existence of five distinct objects o, $o.x$, $o.y$, $o.x.y$ and $o.y.x$ with with five distinct roles P, Q, R, and T (see Figure 6-37). We require Q to be referenced from $P.x$, R to be referenced from $P.y$, T from $Q.y$ and S from $R.x$. In addition to these five roles, we include the roles that ensure that are assigned to the remaining nodes of a graph. We construct these roles to ensure that every model of S_2 contains an object of P role, relying on Property 12.

Finally, we explain how to encode the existence of an accepting Turing machine computation history in the set of role definitions S_1. Let M be a Turing machine and w any input. We use the fact that the computation history of M on input w can be

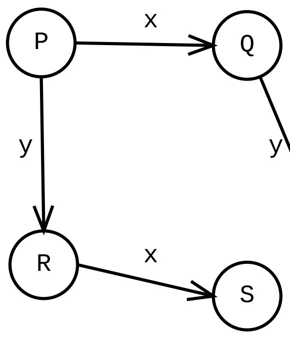


Figure 6-37: Roles that Force Violation of the Commutativity Condition

represented as a matrix, and represent the matrix as a grid. Each row of the matrix represents configuration of the Turing machine encoded as a sequence of symbols. Because all Turing machine transitions change the tape locally, there is a finite set $W_1, \ldots, W_k$ of 3×2 tiles of symbols that characterize the matrix in the following way. We call a 3×2 window in a the matrix *acceptable* if it matches a tile. We use the fact [182] that a matrix represents a computation history of M iff

$$\text{every } 3 \times 2 \text{ window in the matrix is acceptable} \tag{6.4}$$

The condition 6.4 can be split into six conditions $C^{11}, C^{12}, C^{13}, C^{21}, C^{22}, C^{23}$ where C^{ij} ensures that every 3×2 window is acceptable if it starts at (i_1, j_1) where $i_1 \equiv i \pmod 3$ and $j_1 \equiv j \pmod 3$. Let each tile W_t consist of symbols a_t^{11}, a_t^{12}, a_t^{13}, a_t^{21}, a_t^{22}, a_t^{23}.

The set of role definitions S_1 is similar to roles in Example 3 except that it splits the role `A5` into multiple roles. Each new role of S_1 is a sixtuple of positions (t_s, i_s, j_s), where $1 \leq s \leq 6$, such that $a_{t_1}^{i_1 j_1} = a_{t_2}^{i_2 j_2} = \ldots = a_{t_6}^{i_6 j_6}$. Each position (t_s, i_s, j_s) in the role sixtuple ensures that one of the conditions C^{ij} is satisfied where $s = 3(i-1)+j$, using the slot constraints. Along the x field, if $j > 1$, a role with position (t, i, j) as k-th projection can have only aliases from roles with position $(t, i, j-1)$ as k-th projection. If $j = 1$, the aliases can be from roles with $(t', i, 3)$ as the k-th projection. Analogous slot constraints are defined for y fields.

An accepting computation history of the Turing machine M exists iff there exists a matrix where all 3×2 windows are valid which in turn holds iff there exists a grid which satisfied the constraints given by role definitions S_1. A graph which satisfies role definitions S_1 is a grid iff it does not satisfy the role definitions S_2; such graph exists iff the models of S_1 are not included in models of S_2. Hence an accepting computation history of the Turing machine M exists iff the models of S_1 are not included in the models of S_2. Since the first question is undecidable, so is the model inclusion question.

6.10 Related Work

In this section we present the relationship of our work with previous approaches to program analysis, checking, and verification. We first compare our work with the typestate systems including alias types [183] and calculus of capabilities [71]. We mention the previous work on aliasing control for object-oriented languages [121] and the use of roles in object-oriented modeling [162] and database programming languages [102]. We compare our role analysis with shape types [96], graph types [152], path matrix analysis [105], and parametric shape analysis [179]. We briefly relate our approach to some other interprocedural analyses and examine our work in the context of program verification.

6.10.1 Typestate Systems

A typestate system for statically verifying initialization properties of values was proposed in [188, 187]. The type state checking was based on a linear two-pass typestate checking algorithm. In this typestate system, the state of an object depends only on its initialization status. This system did not support aliasing of dynamically allocated structures. Aliasing causes problems for typestate-based systems because the declared typestates of all aliases must change whenever the state of the referred object changes. Faced with the complexity of aliasing, [188] resorted to a more controlled language model based on relations. Requiring the relations to exist only between fully initialized objects enables verification of initialization status of objects in the presence of dynamically growing structures. However, this solution is entirely inadequate for the properties which our role system verifies. Our goal is to verify application-specific properties of objects, and not object initialization. Different objects stored in dynamically growing data structures have different application-specific properties, which our system captures as different roles. When object's properties change, our system verifies that the change is consistent with all relations in which the object participates. Our technique is applicable regardless of whether the relations between objects are implemented as pointer fields of records or in some other way. The data-flow analysis [177] performs verification of constraints on relations and sets that implement dynamic structures, but it does not perform instantiation operation like [179] and our role analysis, which leads to the loss of precision when analyzing destructive updates to data structures.

More recently proposed typestate approaches [74, 200, 183, 71] use linear types to support state changes of dynamically allocated objects. The goal of these systems is to enforce safety properties of low-level code, in particular memory management. This is in contrast with our system which aims at verifying higher-level constraints in a language with a garbage collected heap memory model. The capability calculus [71] allows tracking the aliasing of memory regions by doing a form of compile-time reference counting, but does not track aliasing properties of individual objects. Alias types [183] represent precisely the aliasing of individual objects referenced by local variables, but do not support recursive data structures. Recursive alias types [200] allow specification of recursive data structures as unfolding of basic elaboration steps.

This allows descriptions of tree-like data structures with parent pointers, but does not permit approximating arbitrary data structures. This property of recursive alias types is shared with shape types [96] and graph types [133] discussed below. Another difference compared to our work is that these type systems present only a *type checking*, and not a *type inference* algorithm, whereas our analysis performs role inference inside each procedure. The application of these type systems to an imperative programming language Vault is presented in [74]. Because it is based on alias types and capability calculus, Vault's type system cannot approximate arbitrary data structures. The type system of Vault tracks run-time resources using unique *keys*. To simplify the type checking, Vault requires the equality of sets of keys at each program point. This is in contrast to predicative data-flow analyses such as role analysis, which track the sets of possible aliasing relationships at each program point. Our approach makes the results of the analysis less sensitive to semantic preserving rearrangements of statements in the program.

Like [206, 207], our role analysis performs non-local inference of program properties including the synthesis of loop invariants. The difference is that [206, 207] focus on linear constraints between integers and handle recursive data structures conservatively, whereas we do not handle integer arithmetic but have a more precise representation of the heap that captures the constraints between objects participating in multiple data structures.

6.10.2 Roles in Object-Oriented Programming

It is widely recognized that conventional mechanisms in object-oriented programming languages do not provide sufficient control over object aliasing. As a result, it is not possible to prevent *representation exposure* [79] for linked data structures. As some previous systems, our roles can be used to avoid representation exposure, even though this is not the only purpose of roles.

Islands [121] were designed to help reasoning about object-oriented programs. An island is a set of objects dominated by a *bridge* object in the graph representing the heap. To keep track of aliasing, [121] introduces *unique* and *free* variables with reference counts zero and one, respectively. It also defines a destructive read operation which can be used to pass free objects into procedures. Roles can also be used to enforce the invariant that an object dominates a set of objects reachable along a given set of fields by specifying slot constraints that prevent aliases from objects outside the data structure. Our slot constraints substantially generalize unique and free variables. Our role analysis uses precise shape analysis techniques, which is in sharp contrast with purely syntactic rules of [121].

Balloon types [14] is another system that supports encapsulation. It requires minimal program annotations. The encapsulation in balloon types is enforced using abstract interpretation. The analysis representation records reachability status between objects referenced by variables and relationship of these objects with *clusters* of objects. In most cases our role analysis is more precise than [14] because we track the aliasing properties of objects in recursive data structures, and not only properties of paths between objects.

Ownership types [63, 155] introduce the notion of object ownership to prevent representation exposure. In contrast to the type system [63] where the owner of an object is fixed, our role analysis allows the objects to change the data structure. Furthermore, an object in our system can be simultaneously a member of multiple data structures, and the role analysis verifies the movements of objects specified in procedure interfaces.

The object-oriented community has also become aware of the benefits of the systems where the class of an object changes over the course of the computation. Predicate classes [54] describe objects whose class depends on values of arbitrary predicates. The system [54] computes the values of predicates at run-time and does not attempt to statically infer values of these predicates, leaving to the user even the responsibility of ensuring the disjointness of predicates for incomparable classes. One of the features of predicate classes is a dynamic dispatch based on the current class of the object. In contrast, we are proposing a a selected family of heap constraints and a static role analysis that keeps track of these constraints. Our role system does not have dynamic dispatch. Instead, the declared roles of parameters define a precondition on a procedure call. This precondition changes the operations applicable for an object based on the statically computable information about the dynamic state of the object. Finally, [54] does not attempt to define the state of an object based on object's aliases, which is the central idea of our approach. Even with the great freedom gained by giving up the static checking of classes, systems like [54] cannot verify invariants expressed with our slot constraints; this would in general require adding additional instrumentation fields that track the inverse references.

Dynamic object re-classification [84] presents a system closer to the conventional class-based languages, with method invocation implemented through double dynamic dispatch. The proposal [84] does not statically analyze heap constraints. The work [208] describes a system inspired by a knowledge based reasoning system. The object re-classification in [208] is also implemented by the run-time system. Other approaches propose using design patterns to overcome the absence of language support for dynamically changing classes [98, 94, 109, 197].

The term "role" as used in object-oriented modeling and object-oriented database communities is different from our concept of roles. A role of an object in these systems does not capture object's aliasing properties and other heap constraints. In [162], role denotes the purpose of an object in a collaboration [197] or a design pattern. Our concept of roles captures the associations between objects in a pattern by specifying references that originate or terminate at that object. As in our system, the role of an object in [162] changes over time, and an objects can play multiple roles simultaneously, which corresponds to our partial roles. Our role system ensures the conformance of these design concepts with the actual implementation, improving the reliability of the application. In the database programming language Fibonacci [102, 11] each object plays multiple roles simultaneously. The interface of an object depends on the role through which the object is accessed. This is in contrast to our role system where the role is a structural property of an object. As in most other database implementations, the system [11] checks the inclusion and cardinality constraints on associations at run-time, unlike our static analysis.

6.10.3 Shape Analysis

The precision of our role analysis for tracking references between heap objects is closest to the precision of the shape analysis and verification techniques such as [179, 96, 133, 105]. Whereas these systems focus on analyzing a single data structure, our goal is to analyze interactions between multiple data structures. This is reflected in our choice of the properties to analyze. In particular, the slot constraints tracked by our role analysis are a natural generalization of the sharing predicate in [179] and can be used both to refine the descriptions of data structure nodes and to specify the membership of objects in multiple data structures.

Shape Types [96] is a system for ensuring that the program heap conforms to a context-free graph grammar [87, 171]. As a graph description formalism, context-free graph grammars are incomparable to roles. On the one hand, graph grammars cannot describe an approximation of sparse matrices or specify participation of objects in multiple data structures. On the other hand, the nonparametrized role system presented in this chapter does not include constraints such as "a node must have a self loop". We could express such constraints using roles parametrized by objects. The problem of temporary violations of heap invariants is circumvented in [96] by using high-level graph rewrite rules called *reactions* [97] as part of the implementation language. The model [96] does not support nested reactions on the same data structure or procedure calls from reactions. In contrast, the model of onstage and offstage nodes can be directly applied to a Java-like language, and gives more flexibility to the programmer because roles can be violated in one part of data structure while invoking a procedure on disjoint part of the same data structure. There is no support for procedure specifications in [96]. While simple procedures might be described precisely as reactions, for larger procedures it is necessary to use approximations to keep procedure summaries concise. Our system achieves this goal by using effects as nodeterministic procedure specifications that enable compositional interprocedural analysis.

Graph types and the pointer assertion logic [133, 131, 152] are heap invariant description languages based on monadic second-order logic [193, 69, 134]. In these systems, each graph type data structure must be represented as a spanning tree with additional pointer fields [152] constrained to denote exactly one target node. If a data structure is expressible in this way, the system [152] can verify strong properties about it, an example is manipulation of a threaded tree. Because of constraints on pointer fields, however, it is not possible to approximate data structures such as trees with a pointer to the last accessed leaf, skip lists, or sparse matrices. This restriction also makes it impossible to describe objects that move between data structures while being members of multiple data structures simultaneously. The moving objects cannot be made part of any backbone because their membership in data structures changes over time. The verification of programs in [152] is based on loop invariants. This makes the technique naturally modular and hence no special mechanism is needed for interprocedural analysis. Because the logic is second order, the effects of the procedure can be specified by referring to the sets of nodes affected by the procedure. The problem with this approach is the complexity of loop invariants that describe the intermediate referencing relationships. In contrast, our role analysis uses fixpoint computation to effectively infer loop invariants in the form of sets of role graphs and uses procedures as a unit of a compositional interprocedural analysis.

Like shape analysis techniques [56, 105, 178, 179], we have adopted a constraint-based approach for describing the heap. The constraint based approach allows us to handle a wider range of data structures while potentially giving up some precision.

The path matrix approaches [106, 105] have been used to implement efficient interprocedural analyses that infer one level of referencing relationships, but are not sufficiently precise to track must aliases of heap objects for programs with destructive updates of more complex data structures.

The ADDS data structure description language [124] uses declarations of unique pointers and independent data structure *dimensions* to communicate data structures invariants. Later systems [125, 120] replace these constraints with reachability axioms. None of these systems has a concept of a role which depends on aliasing of an object from other objects. These systems use sound techniques to apply the data structure invariants for parallelization and general dependence testing but do not verify that the data structure invariants are preserved by destructive updates of data structures [123].

The use of the instantiation relation in role analysis is analogous to the materialization operation of [178, 179]. The shape analysis [178, 179] uses abstract interpretation [70] to compute the invariants that the program satisfies at each program point. The values of invariants are stored as 3-valued models for the user-supplied instrumentation predicates. In contrast, our analysis representation is designed to verify a particular role programming model with onstage and offstage nodes. Role graphs use "may" interpretation of edges for offstage nodes and "must" interpretation of edges adjacent to onstage nodes. The abstraction relation is based on graph homomorphism and it is not necessarily a function, so there is no unique best abstract transformer as in the abstract interpretation frameworks. Our role analysis can thus create the summary nodes with different reachability predicates on demand, depending on the behavior of the program. Next, the possibility of having multiple role assignments with static analysis based on the instrumented semantics allows us to capture certain properties of objects that depend not only on the current state of the heap but also on the computation history. Reachability properties in our role analysis are derived from the role graph instead of being explicitly stored as instrumentation predicates. The advantage of our approach is that it naturally handles a class of reachability predicates, without requiring predicate update formulae. Our approach thus avoids the danger of a developer supplying incorrect predicate update formulae and thereby compromising the soundness of the analysis. A disadvantage of our approach is that it does not give must reachability information for paths containing several types of fields where nodes have multiple aliases from those fields. The reason why we *can* recover reachability for e.g. tree-like data structures is that the slot constraint in a role which labels a summary node guarantees the existence of the parent for each node in the path. Our role analysis handles acyclicity by using roles to store the acyclicity assumptions for nodes in recursive data structures. Acyclicity assumptions are instantiated using the the split operation. Our split operation achieves a similar goal to the focus operation of [179]. However, the generic focus algorithm of [145] cannot handle the reachability predicate which is needed for our split operation. This is because it conservatively refuses to focus on edges between two summary nodes to avoid generating an infinite number of structures. Rather than requiring definite values for reachability predicate, our role analysis splits according to reachability properties in the abstract role graph, which illustrates the flexibility of the homomorphism-based abstraction relation.

Type inference algorithms for dynamically typed functional languages [10, 53] have the ability to statically approximate the values of types in higher order languages. These systems usually work with purely functional subsets of functional languages and do not consider the issues of aliasing.

6.10.4 Interprocedural Analyses

A precise interprocedural analysis [168] extends the shape analysis techniques to treat activation records as dynamically allocated structures. The approach also effectively synthesizes an application-specific set of contexts. Our approach differs in that it uses a less precise but more scalable treatment of procedures. It also uses a compositional approach that analyzes each procedure once to verify that it conforms to its specification.

Interprocedural context-sensitive pointer analyses [204, 107, 57] typically compute points-to relationships by caching generated contexts and using fixpoint computation inside strongly connected components of the call graph. Because our analysis tracks more detailed information about the heap, we have chosen to make it compositional at the level of procedures. Our analysis achieves compositionality using procedure effects, which are also useful documentation for the procedure. Like [207] our interprocedural analysis can apply both may and must effects, but our contexts are general graphs with summary nodes and not trees.

The system [116] introduces an annotation language for optimizing libraries. The language describes procedure interfaces which enable optimization of programs that use matrix operations. The supplied function annotations are not verified for the conformance with procedure implementations. In contrast, our goal is to analyze linked data structures to verify heap invariants; it is therefore essential that our role analysis uses sound techniques for both effect verification and effect instantiation.

Our effects are more specific and precise than effects in [132]; as a result they are not commutative. Both verification and instantiation of our effects require specific techniques that precisely keep track of the correspondence between the initial heap of a procedure and the heap at each program point. Our effect application rules implement a form of effect masking. If there are no write effects with the NEW as a target and the source other than NEW, the role graphs in the caller will not be affected.

6.10.5 Program Verification

We can view our role analysis as one component of a general program verification system. The role analysis conservatively attempts to establish a specific class of heap invariants, but does not track other program properties. Verifying data structure invariants is important because the knowledge of these invariants is crucial for reasoning about the behavior of programs with dynamically allocated data structures, which is generally considered difficult. The difficulty of reasoning with dynamically allocated data structures is indicated by some existing systems that verify properties of interfaces but lack automatic verification of conformance between interface and implementation [114], and systems that give up soundness [90, 79]. Advances in reasoning about linked data structures [165, 126] might be a useful starting point for verification tools, although efficient manipulation of properties in verification tools results in different representation requirements than manual reasoning. A combination of model checking [122] and sound automatic model extraction [28] might be an appropriate implementation technique for verifying program properties, but the applicability of this approach for verifying heap invariants remains to be proven.

6.11 Conclusion

We proposed two key ideas: aliasing relationships should determine, in large part, the state of each object, and the type system should use the resulting object states as its fundamental abstraction for describing procedure interfaces and object referencing relationships. We presented a role system that realizes these two key ideas, and described an analysis algorithm that can verify that the program correctly respects the constraints of this role system. The result is that programmers can use roles for a variety of purposes: to ensure the correctness of extended procedure interfaces that take the roles of parameters into account, to verify important data structure consistency properties, to express how procedures move objects between data structures, and to check that the program correctly implements correlated relationships between the states of multiple objects. We therefore expect roles to improve the reliability of the program and its transparency to developers and maintainers. By ensuring that the program conforms to the design constraints expressed in role definitions, role analysis makes design information available to the compilation framework. This enables a range of high-level program transformations such as automatic distribution, parallelization, and memory management.

Chapter 7

An Implementation of Scoped Memory for Real-Time Java

7.1 Introduction

Java is a relatively new and popular programming language. It provides a safe, garbage-collected memory model (no dangling references, buffer overruns, or memory leaks) and enjoys broad support in industry. The goal of the Real-Time Specification for Java [38] is to extend Java to support key features required for writing real-time programs. These features include support for real-time scheduling and predictable memory management.

This paper presents our experience implementing the Real-Time Java memory management extensions. The goal of these extensions is to preserve the safety of the base Java memory model while giving the real-time programmer the additional control that he or she needs to develop programs with predictable memory system behavior. In the base Java memory model, all objects are allocated out of a single garbage-collected heap, raising the issues of garbage-collection pauses and unbounded object allocation times.

Real-Time Java extends this memory model to support two new kinds of memory: *immortal memory* and *scoped memory*. Objects allocated in immortal memory live for the entire execution of the program. The garbage collector scans objects allocated in immortal memory to find (and potentially change) references into the garbage collected heap but does not otherwise manipulate these objects.

Each scoped memory conceptually contains a preallocated region of memory that threads can enter and exit. Once a thread enters a scoped memory, it can allocate objects out of that memory, with each allocation taking a predictable amount of time. When the thread exits the scoped memory, the implementation deallocates all objects allocated in the scoped memory without garbage collection. The specification supports nested entry and exit of scoped memories, which threads can use to obtain a stack of active scoped memories. The lifetimes of the objects stored in the inner scoped memories are contained in the lifetimes of the objects stored in the outer scoped memories. As for objects allocated in immortal memory, the garbage collector scans objects allocated in scoped memory to find (and potentially change) references into the garbage collected heap but does not otherwise manipulate these objects.

The Real-Time Java specification uses dynamic access checks to prevent dangling references and ensure the safety of using scoped memories. If the program attempts to create either 1) a reference from an object allocated in the heap to an object allocated in a scoped memory or 2) a reference from an object allocated in an outer scoped memory to an object allocated in an inner scoped memory, the specification requires the implementation to throw an exception.

7.1.1 Threads and Garbage Collection

The Real-Time Java thread and memory management models are tightly intertwined. Because the garbage collector may temporarily violate key heap invariants, it must be able to suspend any thread that may interact in any way with objects allocated in the garbage-collected heap. Real-Time Java therefore supports two kinds of threads: real-time threads, which may access and refer to objects stored in the garbage-collected heap, and no-heap real-time threads, which may not access or refer to these objects. No-heap real-time threads execute asynchronously with the garbage collector; in particular, they may execute concurrently with or suspend the garbage collector at any time. On the other hand, the garbage collector may suspend real-time threads at any time and for unpredictable lengths of time.

The Real-Time Java specification uses dynamic heap checks to prevent interactions between the garbage collector and no-heap real-time threads. If a no-heap real-time thread attempts to manipulate a reference to an object stored in the garbage-collected heap, the specification requires the implementation to throw an exception. We interpret the term "manipulate" to mean read or write a memory location containing a reference to an object stored in the garbage collected heap, or to execute a method with such a reference passed as a parameter.

7.1.2 Implementation

The primary complication in the implementation is potential interactions between no-heap real-time threads and the garbage collector. One of the basic design goals in the Real-Time Java specification is that the presence of garbage collection should never affect the ability of the no-heap real-time thread to run. We devoted a significant amount of time and energy working with our design to convince ourselves that the interactions did in fact operate in conformance with the specification.

7.1.3 Debugging

We found it difficult to use scoped and immortal memories correctly, especially in the presence of the standard Java libraries, which were not designed with the Real-Time Specification for Java in mind. We therefore found it useful to develop some debugging tools. These tools included a static analysis which finds incorrect uses of scoped memories and a dynamic instrumentation system that enabled the implementation to print out information about the sources of dynamic check failures.

7.2 Programming Model

Because of the proliferation of different kinds of memory areas and threads, Real-Time Java has a fairly complicated programming model.

7.2.1 Entering and Exiting Memory Areas

Real-Time Java provides several kinds of memory areas: scoped memory, immortal memory, and heap memory. Each thread maintains a stack of memory areas; the memory area on the top of the stack is the thread's default memory area. When the thread creates a new object, it is allocated in the default memory area unless the thread explicitly specifies that the object should be allocated in some other memory area. If a thread uses this mechanism to attempt to allocate an object in a scoped memory, the scoped memory must be present in the thread's stack of memory areas. No such restriction exists for objects allocated in immortal or heap memory.

Threads can enter and exit memory areas. When a thread enters a memory area, it pushes the area onto its stack. When it exits the memory area, it pops the area from the stack. There are two ways to enter a memory area: start a parallel thread whose initial stack contains the memory area, or sequentially execute a run method that executes in the memory area. The thread exits the memory area when the run method returns.

The programming model is complicated somewhat by the fact that 1) a single thread can reenter a memory area multiple times, and 2) different threads can enter memory areas in different orders. Assume, for example, that we have two scoped memories A and B and two threads T and S. T can first enter A, then B, then A again, while S can first enter B, then A, then B again. The objects in A and B are deallocated only when T exits A, then B, then A again, and S exits B, then A, then B again. Note that even though the programming model specifies nested entry and exit of memory areas, these nested entries and exits do not directly translate into a hierarchical inclusion relationship between the lifetimes of different memory areas.

7.2.2 Scoped Memories

Scoped memories, in effect, provide a form of region-based memory allocation. They differ somewhat from other forms of region-based memory allocation [100] in that each scoped memory is associated with one or more computations (each computation is typically a thread, but can also be the execution of a sequentially invoked run method), with all of the objects in the scoped memory deallocated when all of its associated computations terminate.

The primary issue with scoped memories is ensuring that their use does not create dangling references, which are references to objects allocated in scoped memories that have been deallocated. The basic strategy is to use dynamic access checks to prevent the program from creating a reference to an object in a scoped memory from an object allocated in either heap memory, immortal memory, or a scoped memory whose lifetime encloses that of the first scoped memory. Whenever a thread attempts to store a reference to a first object into a field in a second object, an access check verifies that:

> If the first object is allocated in a scoped memory, then the second object must also be allocated in a scoped memory whose lifetime is contained in the lifetime of the scoped memory containing the first object.

The implementation checks the containment by looking at the thread's stack of scoped memories and checking that either 1) the objects are allocated in the same scoped memory, or 2) the thread first entered the scoped memory of the second object before it first entered the scoped memory of the first object. If this check fails, the implementation throws an exception.

Let's consider a quick example to clarify the situation. Assume we have two scoped memories A and B, two objects O and P, with O allocated in A and P allocated in B, and two threads T and S. Also assume that T first enters A, then B, then A again, while S first enters B, then A, then B again. Now T can store a reference to O in a field of P, but cannot store a reference to P in a field of O. For S, the situation is reversed: S cannot store a reference to O in a field of P, but can store a reference to P in a field of O.

7.2.3 No-Heap Real-Time Threads

No-heap real-time threads have an additional set of restrictions; these restrictions are intended to ensure that the thread does not interfere with the garbage collector. Specifically, the Real-Time Specification for Java states that a no-heap real-time thread, which can run asynchronously with the garbage collector, "is never allowed to allocate or reference any object allocated in the heap nor is it even allowed to manipulate the references to objects in the heap." Our implementation uses five runtime heap checks to ensure that a no-heap real-time thread does not interfere with garbage collection by manipulating heap references. The implementation uses three of these types of checks, **CALL**, **METHOD**, and **NATIVECALL** to guard against poorly implemented native methods or illegal compiler calls into the runtime. These three checks can be removed if all native and runtime code is known to operate correctly.

- **CALL**: A native method invoked by a no-heap real-time thread cannot return a reference to a heap allocated object.

- **METHOD**: A Java method cannot be passed a heap allocated object as an argument while running in a no-heap real-time thread.

- **NATIVECALL**: A compiler-generated call into the runtime implementation from a no-heap real-time thread cannot return a reference to a heap allocated object.

- **READ**: A no-heap real-time thread cannot read a reference to a heap allocated object.

- **WRITE**: As part of the execution of an assignment statement, a no-heap real-time thread cannot overwrite a reference to a heap allocated object.

7.3 Example

We next present an example that illustrates some of the features of the Real-Time Specification for Java. Figure 7-1 presents a sample program written in Real-Time Java. This program is a version of the familiar "Hello World" program augmented to use the Real-Time Java features. It first creates a scoped memory with a worst-case Linear Time allocation scheme (`LTMemory`) with a size of 1000 bytes. It then runs the code of the `run` method in this new scope. The `run` method creates a new variable time allocation scoped memory (the `VTMemory` object) and a new `Worker` `NoHeapRealtimeThread`. Both of these objects are allocated in the `LTMemory` scoped memory. The `run` method then starts the `Worker` thread and executes its `join` method, which will return when the `Worker` finishes.

The `Worker` thread runs in the new `VTMemory`. The `Worker`'s `run` method allocates a new `String`[1] in `ImmortalMemory` and stores a reference to this string in the static `results` field of the `Main` class, which was previously initialized to `null`. The `Worker` then creates a new `String`, "Hello World!", to place in the array. The worker then finishes, and the implementation deallocates all of the objects allocated in the `VTMemory`. Back in the main thread, the `join` method returns, and the main thread returns back out of its `run` method. The implementation deallocates all of the objects allocated in the `LTMemory`. Finally, the main thread prints "Hello World", the first element of the `results` array, to the screen.

Note that the `LTMemory` and `VTMemory` constructors differ slightly from the constructors described in the Realtime Java specification. We implemented these constructors in addition to the specified constructors to provide additional flexibility and convenience for the programmer.

This Hello World program is a legal program using our system. However, any of the following changes would make it an illegal program:

1. Replace the `im.newInstance...` with `‘‘Hello World!’’` and there would be an illegal reference from an `ImmortalMemory` to a `ScopedMemory`.

2. Replace the `im.newArray...` with `new String[1]` and there would be an illegal static reference to a `ScopedMemory`.

```
class Worker extends NoHeapRealtimeThread {
  Worker(MemoryArea ma) { super(ma); }
  public void run() {
    ImmortalMemory im = ImmortalMemory.instance();
    try {
      Main.results =
          (String[]) im.newArray(String.class, new int[] { 1 });
      Main.results[0] =
          (String)im.newInstance(String.class,
                                 new Class[] { String.class },
                                 new Object[] { ‘‘Hello World!’’ });
    } catch (Exception e) { System.exit(-1); }
  }
}
public class Main {
  public static String[] results = null;
  public static void main(String args[]) {
    LTMemory lt = new LTMemory(1000);
    lt.enter(new Runnable() {
      public void run() {
        Worker w = new Worker(new VTMemory());
        w.start();
        try { w.join(); }
        catch (Exception e) { System.out.println(e); }
      }
    });
    System.out.println(results[0]);
  }
}
```

Figure 7-1: A Real-Time Java Example Program

3. Replace the `ImmortalMemory.instance()` with `HeapMemory.instance()`, and there would be an illegal heap reference in a `NoHeapRealtimeThread` (**READ**).

4. Replace the `null` with a `new String[1]` and the `NoHeapRealtimeThread` would be illegally destroying a heap reference by assigning `Main.results` (**WRITE**).

5. Place the `Worker w` in the `main` method and the assignment `w = new Worker...` would illegally create a reference from the heap to a `ScopedMemory`.

6. Place the `System.out` in the `NoHeapRealtimeThread` and the `NoHeapRealtimeThread` would be illegally reading from the heap. `System.out` is initialized in the initial `MemoryArea` at the start of the program, the `HeapMemory` (**READ**) As a consequence, the `NoHeapRealtimeThread` cannot `System.out.println` the message from the exception.

7. Place the entire `Worker w = new Worker(new VTMemory());` outside the `LTMemory`

scope, and the `this` pointer of the `NoHeapRealtimeThread` would illegally point to the heap (**METHOD**).

7.4 Implementation

Our discussion of the implementation focuses on three aspects: implementing the heap and access checks, implementing the additional scoped immortal memory functionality, and ensuring the absence of interactions between no-heap real-time threads and the garbage collector.

7.4.1 Heap Check Implementation

The implementation must be able to take an arbitrary reference to an object and determine the kind of memory area in which it is allocated. To support this functionality, our implementation adds an extra field to the header of each object. This field contains a pointer to the memory area in which the object is allocated.

One complication with this scheme is that the garbage collector may violate object representation invariants during collection. If a no-heap real-time thread attempts to use the field in the object header to determine if an object is allocated in the heap, it may access memory rendered invalid by the actions of the garbage collector. We therefore need a mechanism which enables a no-heap real-time thread to differentiate between heap references and other references without attempting to access the memory area field of the object.

We first considered allocating a contiguous address region for the heap, then checking to see if the reference falls within this region. We decided not to use this approach because of potential interactions between the garbage collector and the code in the no-heap real-time thread that checks if the reference falls within the heap. Specifically, using this scheme would force the garbage collector to always maintain the invariant that the current heap address region include all previous heap address regions. We were unwilling to impose this restriction on the collector.

We then considered a variety of other schemes, but eventually settled on the (relatively simple) approach of setting the low bit of all heap references. The generated code masks off this bit before dereferencing the pointer to access the object. With this approach, no-heap real-time threads can simply check the low bit of each reference to check if the reference points into the heap or not.

Our current system uses the memory area field in the object header to obtain information about objects allocated in scoped memories and immortal memory. The basic assumption is that the objects allocated in these kinds of memory areas will never move or have their memory area field temporarily corrupted or invalidated.

Figure 7-2 presents the code that the compiler emits for each heap check; Figure 7-3 presents the code that determines if the current thread is a no-heap real-time thread. Note that the emitted code first checks to see if the reference is a heap reference — our expectation is that most Real-Time Java programs will manipulate relatively

READ	WRITE	CALL

use of `*refExp` in `exp`
 	
becomes:
 	
`heapRef = *refExp;`
`if (heapRef&1)`
` heapCheck(heapRef);`
`[*heapRef/*refExp] exp`
	`*refExp = exp;`
	
	becomes:
	
	`heapRef = *refExp;`
	`if (heapRef&1)`
	` heapCheck(heapRef);`
	`refExp = exp;`
		`refExp = call(args);`
		
		becomes:
		
		`heapRef = call(args);`
		`if (heapRef&1)`
		` heapCheck(heapRef);`
		`refExp = heapRef;`

NATIVECALL	METHOD

`refExp = nativecall(args);`
 	
becomes:
 	
`heapRef = nativecall(args);`
`if (heapRef&1)`
` heapCheck(heapRef);`
`refExp = heapRef;`
	`method(args) { body }`
	
	becomes:
	
	`method(args) {`
	` for arg in args:`
	`  if (arg&1)`
	`   heapCheck(arg);`
	`body }`

Figure 7-2: Emitted Code For Heap Checks

few references to heap-allocated objects. This expectation holds for our benchmark programs (see Section 7.6).

7.4.2 Access Check Implementation

The access checks must be able to determine if the lifetime of a scoped memory area A is included in the lifetime of another scoped memory area B. The implementation searches the thread's stack of memory areas to perform this check. It first searches for the occurrence of A closest to the start of the stack (recall that A may occur multiple times on the stack). It then searches to check if there is an occurrence of B between that occurrence of A and the start of the stack. If so, the access check succeeds; otherwise, it fails.

The current implementation optimizes this check by first checking to see if A and B are the same scoped memory area. Figure 7-4 presents the emitted code for the access checks, while Figure 7-5 presents some of the run-time code that this emitted code invokes.

7.4.3 Operations on Memory Areas

The implementation needs to perform three basic operations on scoped and immortal memory areas: allocate an object in the area, deallocate all objects in the area, and provide the garbage collector with the set of all heap references stored in the memory area. Note a potential interaction between the garbage collector and no-heap real-time

```
#ifdef DEBUG
  void heapCheck(unwrapped_jobject* heapRef, const int source_line,
                 const char* source_fileName, const char* operation) {
#else       /* operation = READ, WRITE, CALL, NATIVECALL, or METHOD */
  void heapCheck(unwrapped_jobject* heapRef) {
#endif
    JNIEnv* env = FNI_GetJNIEnv();
    /* determine if in a NoHeapRealtimeThread */
    if (((struct FNI_Thread_State*)env)->noheap) {
       /* optionally print helpful debugging info */
       /* throw exception */
    }
  }
```

Figure 7-3: The `heapCheck` function

New Object (or Array):

```
  obj = new foo(); (or obj = new foo()[1][2][3];)
```

becomes:

```
  ma = RealtimeThread.currentRealtimeThread().getMemoryArea();
  obj = new foo(); (or obj = new foo()[1][2][3];)
  obj.memoryArea = ma;
```

Access check:

```
  obj.foo = bar;
```

becomes:

```
  ma = MemoryArea.getMemoryArea(obj); // or ma = ImmortalMemory.instance(),
  ma.checkAccess(bar);                //   if a static field)
  obj.foo = bar;
```

Figure 7-4: Emitted Code for Access Checks

In `MemoryArea`:

```
public void checkAccess(Object obj) {
  if ((obj != null) && (obj.memoryArea != null) && obj.memoryArea.scoped) {
    /* Helpful native method prints out all debugging info. */
    throwIllegalAssignmentError(obj, obj.memoryArea);
  }
}
```

Overridden in `ScopedMemory`:

```
public void checkAccess(Object obj) {
  if (obj != null) {
    MemoryArea target = getMemoryArea(obj);
    if ((this != target) && target.scoped &&
        (!RealtimeThread.currentRealtimeThread()
         .checkAccess(this, target))) {
        throwIllegalAssignmentError(obj, target);
    }
  }
}
```

In `RealtimeThread`:

```
boolean checkAccess(MemoryArea source, MemoryArea target) {
  MemBlockStack sourceStack = (source == getMemoryArea()) ?
    memBlockStack : memBlockStack.first(source);
  return (sourceStack != null) && (sourceStack.first(target) != null);
}
```

Figure 7-5: Code for performing access checks

threads. The garbage collector may be in the process of retrieving the heap references stored in a memory area when a no-heap real-time thread (operating concurrently with or interrupting the garbage collector) allocates objects in that memory area. The garbage collector must operate correctly in the face of the resulting changes to the underlying memory area data structures. The system design also cannot involve locks shared between the no-heap real-time thread and the garbage collector (the garbage collector is not allowed to block a no-heap real-time thread). But the garbage collector may assume that the actions of the no-heap real-time thread do not change the set of heap references stored in the memory area.

Each memory area may have its own object allocation algorithm. Because the same code may execute in different memory areas at different times, our implementation is set up to dynamically determine the allocation algorithm to use based on the current memory area. Whenever a thread allocates an object, it looks up a data structure associated with the memory area. A field in this structure contains a pointer to the allocation function to invoke. This structure also contains a pointer to a function that retrieves all of the heap references from the area, and a function that deallocates all of the objects allocated in the area.

7.4.4 Memory Area Reference Counts

As described in the Real-Time Java Specification, each memory area maintains a count of the number of threads currently operating within that region. These counts are (atomically) updated when threads enter or exit the region. When the count becomes zero, the implementation deallocates all objects in the area.

Consider the following situation. A thread exits a memory area, causing its reference count to become zero, at which point the implementation starts to invoke finalizers on the objects in the memory area as part of the deallocation process. While the finalizers are running, a no-heap real-time thread enters the memory area. According to the Real-Time Java specification, the no-heap real-time thread blocks until the finalizers finish running. There is no mention of the priority with which the finalizers run, raising the potential issue that the no-heap real-time thread may be arbitrarily delayed. A final problem occurs if the no-heap real-time thread first acquires a lock, a finalizer running in the memory area then attempts to acquire the lock (blocking because the no-heap real-time thread holds the lock), then the no-heap real-time thread attempts to enter the memory area. The result is deadlock — the no-heap real-time thread waits for the finalizer to finish, but the finalizer waits for the no-heap real-time thread to release the lock.

7.4.5 Memory Allocation Algorithms

We have implemented two simple allocators for scoped memory areas: a stack allocator and a `malloc`-based allocator. The current implementation uses the stack allocator for instances of `LTMemory`, which guarantee linear-time allocation, and the `malloc`-based allocator for instances of `VTMemory`, which provide no time guarantees.

The stack allocator starts with a fixed amount of available free memory. It maintains a pointer to the next free address. To allocate a block of memory, it increments the pointer by the size of the block, then returns the old value of the pointer as a reference to the newly allocated block. Our current implementation uses this allocation strategy for instances of the `LTMemory` class, which guarantees a linear time allocation strategy.

There is a complication associated with this implementation. Note that multiple threads can attempt to concurrently allocate memory from the same stack allocator. The implementation must therefore use some mechanism to ensure that the allocations take place atomically. Note that the use of lock synchronization could cause an unfortunate coupling between real-time threads, no-heap real-time threads, and the garbage collector. Consider the following scenario. A real-time thread starts to allocate memory, acquires the lock, is suspended by the garbage collector, which is then suspended by a no-heap real-time thread that also attempts to allocate memory from the same allocator. Unless the implementation does something clever, it could either deadlock or force the no-heap real-time thread to wait until the garbage collector releases the real-time thread to complete its memory allocation.

Our current implementation avoids this problem by using a lock-free, nonblocking atomic exchange-and-add instruction to perform the pointer updates. Note that on an multiprocessor in the presence of contention from multiple threads attempting to concurrently allocate from the same memory allocator, this approach could cause the allocation time to depend on the precise timing behavior of the atomic instructions. We would expect some machines to provide no guarantee at all about the termination time of these instructions.

The `malloc`-based allocator simply calls the standard `malloc` routine to allocate memory. Our implementation uses this strategy for instances of `LTMemory`. To provide the garbage collector with a list of heap references, our implementation keeps a linked list of the allocated memory blocks and can scan these blocks on demand to locate references into the heap.

Our design makes adding a new allocator easy; the `malloc`-based allocator required only 25 lines of C code and only 45 minutes of coding, debugging, and testing time. Although the system is flexible enough to support multiple dynamically-changing allocation routines, `VTMemory`s use the linked-list allocator, while `LTMemory`s use the stack-allocator.

7.4.6 Garbage Collector Interactions

References from heap objects can point both to other heap objects and to objects allocated in immortal memory. The garbage collector must therefore recognize references to immortal memory and treat objects allocated in immortal memory differently than objects allocated in heap memory. In particular, the garbage collector cannot change the objects in ways that that would interact with concurrently executing no-heap real-time threads.

Our implementation handles this issue as follows. The garbage collector first scans the immortal and scoped memories to extract all references from objects allocated in these memories to heap allocated objects. This scan is coded to operate correctly in the presence of concurrent updates from no-heap real-time threads. The garbage collector uses the extracted heap references as part of its root set.

During the collection phase, the collector does not trace references to objects allocated in immortal memory. If the collector moves objects, it may need to update references from objects allocated in immortal memory or scoped memories to objects allocated in the heap. It performs these updates in such a way that it does not interfere with the ability of no-heap real-time threads to recognize such references as referring to objects allocated in the heap. Note that because no-heap real-time threads may access heap references only to perform heap checks, this property ensures that the garbage collector and no-heap real-time threads do not inappropriately interfere.

7.5 Debugging Real-Time Java Programs

An additional design goal becomes extremely important when actually developing Real-Time Java programs: ease of debugging. During the development process, facilitating debugging became a primary design goal. In fact, we found it close to impossible to develop error-free Real-Time Java programs without some sort of assistance (either a debugging system or static analysis) that helped us locate the reason for our problems using the different kinds of memory areas. Our debugging was especially complicated by the fact that the standard Java libraries basically don't work at all with no-heap real-time threads.

7.5.1 Incremental Debugging

During our development of Real-Time Java programs, we found the following incremental debugging strategy to be useful. We first stubbed out all of the Real-Time Java heap and access checks and special memory allocation strategies, in effect running the Real-Time Java program as a standard Java program. We used this version to debug the basic functionality of the program. We then added the heap and access checks, and used this version to debug the memory allocation strategy of the program. We were able to use this strategy to divide the debugging process into stages, with a manageable amount of bugs found at each stage.

It is also possible to use static analysis to verify the correct use of Real-Time Java scoped memories [191]. We had access to such an analysis when we were implementing our benchmark programs, and the analysis was very useful for helping us debug our use of scoped memories. It also dramatically increased our confidence in the correctness of the final program, and enabled a static check elimination optimization that improved the performance of the program.

7.5.2 Additional Runtime Debugging Information

Heap and access checks can be used to help detect mistakes early in the development process, but additional tools may be necessary to understand and fix those mistakes in a timely fashion. We therefore augmented the memory area data structure to produce a debugging system that helps programmers understand the causes of object referencing errors.

When a debugging flag is enabled, the implementation attaches the original Java source code file name and line number to each allocated object. Furthermore, with the use of macros, we also obtain allocation site information for native methods. We store this allocation site information in a list associated with the memory area in which the object is allocated. Given any arbitrary object reference, a debugging function can retrieve the debugging information for the object. Combined with a stack trace at the point of an illegal assignment or reference, the allocation site information from both the source and destination of an illegal assignment or the location of an illegal reference can be instrumental in quickly determining the exact cause of the error and the objects responsible. Allocation site information can also be displayed at the time of allocation to provide a program trace which can help determine control flow, putting the reference in a context at the time of the error.

7.6 Results

We implemented the Real-Time Java memory extensions in the MIT Flex compiler infrastructure.[1] Flex is an ahead-of-time compiler for Java that generates both native code and C; it can use a variety of garbage collectors. For these experiments, we generated C and used the Boehm-Demers-Weiser conservative garbage collector.

We obtained several benchmark programs and used these programs to measure the overhead of the heap checks and access checks. Our benchmarks include Barnes, a hierarchical N-body solver, and Water, which simulates water molecules in the liquid state. Initially these benchmarks allocated all objects in the heap. We modified the benchmarks to use scoped memories whenever possible. We also present results for two synthetic benchmarks, Tree and Array, that use object field assignment heavily. These benchmarks are designed to obtain the maximum possible benefit from heap and access check elimination.

Table 7.1 presents the number of objects we were able to allocate in each of the different kinds of memory areas. The goal is to allocate as many objects as possible in scoped memory areas; the results show that we were able to modify the programs to allocate the vast majority of their objects in scoped memories. Java programs also allocate arrays; Table 7.2 presents the number of arrays that we were able to allocate in scoped memories. As for objects, we were able to allocate the vast majority of arrays in scoped memories.

Table 7.3 presents the number and type of access checks for each benchmark. Recall that there is a check every time the program stores a reference. The different columns of the table break down the checks into categories depending on the target of the store and the memory area that the stored reference refers to. For example, the

[1]Available at `www.flexc.lcs.mit.edu`

Table 7.1: Number of Objects Allocated In Different Memory Areas

Benchmark	Heap	Scoped	Immortal	Total
Array	13	4	0	17
Tree	13	65,534	0	65,547
Water	406,895	3,345,711	0	3,752,606
Barnes	16,058	4,681,708	0	4,697,766

Table 7.2: Number of Arrays Allocated In Different Memory Areas

Benchmark	Heap	Scoped	Immortal	Total
Array	36	4	0	40
Tree	36	0	0	36
Water	405,943	13,160,641	0	13,566,584
Barnes	14,871	4,530,765	0	4,545,636

Table 7.3: Access Check Counts

Benchmark	Heap to Heap	Heap to Immortal	Scoped to Heap	Scoped to Scoped	Scoped to Immortal	Immortal to Heap	Immortal to Immortal
Array	14	8	0	400,040,000	0	0	0
Tree	14	8	0	65,597,532	65,601,536	0	0
Water	409,907	0	17,836	9,890,211	844	3	1
Barnes	90,856	80,448	9,742	4,596,716	1328	0	0

Scoped to Heap column counts the number of times the program stored a reference to heap memory into an object or array allocated in a scoped memory.

Table 7.4 presents the running times of the benchmarks. We report results for six different versions of the program. The first three versions all have both heap and access checks, and vary in the memory area they use for objects that we were able to allocate in scoped memory. The Heap version allocates all objects in the heap. The VT version allocates scoped-memory objects in instances of `VTMemory` (which use `malloc`-based allocation); the LT version allocates scoped-memory objects in instances of `LTMemory` (which use stack-based allocation). The next three versions use the same allocation strategy, but the compiler generates code that omits all of the checks. For our benchmarks, our static analysis is able to verify that none of the checks will fail, enabling the compiler to eliminate all of these checks [191].

These results show that checks add significant overhead for all benchmarks. But the use of scoped memories produces significant performance gains for Barnes and Water. In the end, the use of scoped memories without checks significantly increases the overall performance of the program. To investigate the causes of the performance

Table 7.4: Execution Times of Benchmark Programs

	With Checks			Without Checks		
Benchmark	Heap	VT	LT	Heap	VT	LT
Array	28.1	43.2	43.1	7.8	7.7	8.0
Tree	13.2	16.6	16.6	6.9	6.9	6.9
Water	58.2	47.4	37.8	52.3	40.2	30.2
Barnes	38.3	22.3	17.2	34.7	19.5	14.4

differences, we instrumented the run-time system to measure the garbage collection pause times. Based on these measurements, we attribute most of the performance differences between the versions of Water and Barnes with and without scoped memories to garbage collection overheads. Specifically, the use of scoped memories improved every aspect of the garbage collector: it reduced the total garbage collection overhead, increased the time between collections, and significantly reduced the pause times for each collection.

For Array and Tree, there is almost no garbage collection for any of the versions and the versions without checks all exhibit basically the same performance. With checks, the versions that allocate all objects in the heap run faster than the versions that allocate objects in scoped memories. We attribute this performance difference to the fact that heap to heap access checks are faster than scope to scope access checks.

7.7 Related Work

Christiansen and Velschow suggested a region-based approach to memory management in Java; they called their system RegJava[60]. They found that fixed-size regions have better performance than variable-sized regions and that region allocation has more predictable and often better performance than garbage collection. Static analysis can be used to detect where region annotations should be placed, but the annotations often need to be manually modified for performance reasons. Compiling a subset of Java which did not include threads or exceptions to C++, the RegJava system does not allow regions to coexist with garbage collection. Finally, the RegJava system permits the creation of dangling references.

Gay and Aiken implemented a region-based extension of C called C@ which used reference counting on regions to safely allocate and deallocate regions with a minimum of overhead[99]. Using special region pointers and explicit `deleteregion` calls, Gay and Aiken provide a means of explicitly manipulating region-allocated memory. They found that region-based allocation often uses less memory and is faster than traditional malloc/free-based memory management. Unfortunately, counting escaping references in C@ can incur up to 16% overhead. Both Christiansen and Velschow and Gay and Aiken explore the implications of region allocation for enhancing locality.

Gay and Aiken also produced RC [100], an explicit region allocation dialect of C, and an improvement over C@. RC uses heirarchically structured regions and sameregion, traditional, and parentptr pointer annotations to reduce the reference counting overhead to at most 11% of execution time. Using static analysis to reduce the number of safety checks, RC demonstrates up to a 58% speedup in programs that use regions as opposed to garbage collection or the typical malloc and free. RC uses 8KB aligned pages to allocate memory and the runtime keeps a map of pages to regions to resolve regionof calls quickly. Regions have a partial order to facilitate parentptr checks.

Region analysis seems to work best when the programmer is aware of the analysis, indicating that explicitly defined regions which give the programmer control over storage allocation may lead to more efficient programs. For example, the Tofte/Talpin ML inference system required that the programmer be aware of the analysis to guard against excessive memory leaks [195]. Programs which use regions explicitly may be more hierarchically structured with respect to memory usage by programmer design than programs intended for the traditional, garbage-collected heap. Therefore, Real-Time Java uses hierarchically-structured, explicit, reference-counted regions that strictly prohibit the creation of dangling references.

Our research is distinguished by the fact that Real-Time Java is a strict superset of the Java language; any program written in ordinary Java can run in our Real-Time Java system. Furthermore, a Real-Time Java thread which uses region allocation and/or heap allocation can run concurrently with a thread from any ordinary Java program, and we support several kinds of region-based allocation and allocation in a garbage collected heap in the same system.

7.8 Conclusion

The Real-Time Java Specification promises to bring the benefits of Java to programmers building real-time systems. One of the key aspects of the specification is extending the Java memory model to give the programmer more control over the memory management. We have implemented these extensions. We found that the primary implementation complication was ensuring a lack of interference between the garbage collector and no-heap real-time threads, which execute asynchronously with respect to the design. We also found debugging tools necessary for the effective development of programs that use the Real-Time Java memory management extensions. We used both a static analysis and a dynamic debugging system to help locate the source of incorrect uses of these extensions.

THIS PAGE WAS INTENTIONALLY LEFT BLANK

Chapter 8

Ownership Types for Safe Region-Based Memory Management in Real-Time Java

8.1 Introduction

The Real-Time Specification for Java (RTSJ) [38] provides a framework for building real-time systems. The RTSJ allows a program to create real-time threads with hard real-time constraints. These real-time threads cannot use the garbage-collected heap because they cannot afford to be interrupted for unbounded amounts of time by the garbage collector. Instead, the RTSJ allows these threads to use objects allocated in immortal memory (which is never garbage collected) or in regions [195]. Region-based memory management systems structure memory by grouping objects in regions under program control. Memory is reclaimed by deleting regions, freeing all objects stored therein. The RTSJ uses runtime checks to ensure that deleting a region does not create dangling references and that real-time threads do not access heap references.

This chapter presents a static type system for writing real-time programs in Java. Our system guarantees that the RTSJ runtime checks will never fail for well-typed programs. Our system thus serves as a front-end for the RTSJ platform. It offers two advantages to real-time programmers. First, it provides an important safety guarantee that a program will never fail because of a failed RTSJ runtime check. Second, it allows RTSJ implementations to remove the RTSJ runtime checks and eliminate the associated overhead.

Our approach is applicable even outside the RTSJ context; it could be adapted to provide safe region-based memory management for other real-time languages as well.

Our system makes several important technical contributions over previous type systems for region-based memory management. For object-oriented programs, it combines region types [59, 71, 111, 195] and ownership types [43, 44, 46, 62, 63] in a unified type system framework. Region types statically ensure that programs never follow dangling references. Ownership types statically enforce object encapsulation and enable modular reasoning about program correctness in object-oriented programs.

Consider, for example, a `Stack` object `s` that is implemented using a `Vector` subobject `v`. To reason locally about the correctness of the `Stack` implementation, a programmer must know that `v` is not directly accessed by objects outside `s`. With ownership types, a programmer can declare that `s` *owns* `v`. The type system then statically ensures that `v` is encapsulated within `s`.

In an object-oriented language that only has region types (e.g., [59]), the types of `s` and `v` would declare that they are allocated in some region `r`. In an object-oriented language that only has ownership types, the type of `v` would declare that it is owned by `s`. Our type system provides a simple unified mechanism to declare *both* properties. The type of `s` can declare that it is allocated in `r` and the type of `v` can declare that it is owned by `s`. Our system then statically ensures that both objects are allocated in `r`, that there are no pointers to `v` and `s` after `r` is deleted, and that `v` is encapsulated within `s`. Our system thus combines the benefits of region types and ownership types.

Our system extends region types to multithreaded programs by allowing explicit memory management for objects shared between threads. It allows threads to communicate through objects in *shared regions* in addition to the heap. A shared region is deleted when all threads exit the region. However, programs in a system with only shared regions (e.g., [110]) will have memory leaks if two long-lived threads communicate by creating objects in a shared region. This is because the objects will not be deleted until both threads exit the shared region. To solve this problem, we introduce the notion of *subregions* within a shared region. A subregion can be deleted more frequently, for example, after each loop iteration in the long-lived threads.

Our system also introduces *typed portal fields* in subregions to serve as a starting point for inter-thread communication. Portals also allow typed communication, so threads do not have to downcast from `Object` to more specific types. Our approach therefore avoids any dynamic type errors associated with these downcasts. Our system introduces user-defined *region kinds* to support subregions and portal fields.

Our system extends region types to real-time programs by statically ensuring that real-time threads do not interfere with the garbage collector. Our system augments region kind declarations with *region policy* declarations. It supports two policies for creating regions as in the RTSJ. A region can be an LT (Linear Time) region, or a VT (Variable Time) region. Memory for an LT region is preallocated at region creation time, so allocating an object in an LT region only takes time proportional to the size of the object (because all the bytes have to be zeroed). Memory for a VT region is allocated on demand, so allocating an object in a VT region takes variable time. Our system checks that real-time threads do not use heap references, create new regions, or allocate objects in VT regions.

Our system also prevents an RTSJ *priority inversion* problem. In the RTSJ, any thread entering a region waits if there are threads exiting the region. If a regular thread exiting a region is suspended by the garbage collector, then a real-time thread entering the region might have to wait for an unbounded amount of time. Our type system statically ensures that this priority inversion problem cannot happen.

Finally, we note that ownership-based type systems have also been used for preventing data races [46] and deadlocks [43], for supporting modular software upgrades in persistent object stores [45], for modular specification of effects clauses in the presence of subtyping [44, 46] (so they can be used as an alternative to data groups [144]), and for program understanding [13]. We are currently unifying the type system presented in this chapter with the above type systems [41]. The unified ownership type system requires little programming overhead, its typechecking is fast and scalable, and it provides several benefits. The unified ownership type system thus offers a promising approach for making object-oriented programs more reliable.

Contributions

To summarize, the research presented in this chapter makes the following contributions:

- **Region types for object-oriented programs:** Our system combines region types and ownership types in a unified type system framework that statically enforces object encapsulation as well as enables safe region-based memory management.

- **Region types for multithreaded programs:** Our system introduces 1) *subregions* within a shared region, so that long-lived threads can share objects without using the heap and without memory leaks and 2) *typed portal fields* to serve as a starting point for typed inter-thread communication. It also introduces user-defined *region kinds* to support subregions and portals.

- **Region types for real-time programs:** Our system allows programs to create LT (Linear Time) and VT (Variable Time) regions as in the RTSJ. It checks that real-time threads do not use heap references, create new regions, or allocate objects in VT regions, so that they do not wait for unbounded amounts of time. It also prevents an RTSJ *priority inversion* problem.

- **Type inference:** Our system uses a combination of intra-procedural type inference and well-chosen defaults to significantly reduce programming overhead. Our approach permits separate compilation.

- **Experience:** We have implemented several programs in our system. Our experience indicates that our type system is sufficiently expressive and requires little programming overhead. We also ran the programs on our RTSJ platform [32, 33]. Our experiments show that eliminating the RTSJ runtime checks using a static type system can significantly speed-up programs.

The paper is organized as follows. Section 8.2 describes our type system. Section 8.3 describes our experimental results. Section 8.4 presents related work. Section 8.5 concludes.

O1. The ownership relation forms a forest of trees.

O2. If region $r \succeq_o$ object x, then x is allocated in r.

O3. If object $z \succeq_o y$ but $z \not\succeq_o x$, then x cannot access y.

Figure 8-1: Ownership Properties

8.2 Type System

This section presents our type system for safe region-based memory management. Sections 8.2.1, 8.2.2, and 8.2.3 describe our type system. Section 8.2.4 presents some of the important rules for typechecking. The complete set of rules are presented in [47]. Section 8.2.5 describes type inference techniques. Section 8.2.6 describes how programs written in our system are translated to run on our RTSJ platform.

8.2.1 Regions for Object-Oriented Programs

This section presents our type system for safe region-based memory management in single-threaded object-oriented programs. It combines the benefits of region types [59, 71, 111, 195] and ownership types [43, 44, 46, 62, 63]. Region types statically ensure that programs using region-based memory management are memory-safe, that is, they never follow dangling references. Ownership types statically enforce object encapsulation. The idea is that an object can *own* subobjects that it depends on, thus preventing them from being accessible outside. (An object x *depends on* [143, 44] subobject y if x calls methods of y and furthermore these calls expose mutable behavior of y in a way that affects the invariants of x.) Object encapsulation enables local reasoning about program correctness in object-oriented programs.

Ownership Relation Objects in our system are allocated in regions. Every object has an owner. An object can be owned by another object, or by a region. We write $o_1 \succeq_o o_2$ if o_1 directly or transitively owns o_2 or if o_1 is the same as o_2. The relation $\succeq_o$ is thus the reflexive transitive closure of the *owns* relation. Our type system statically guarantees the properties in Figure 8-1. O1 states that our ownership relation has no cycles. O2 states that if an object is owned by a region, then that object and all its subobjects are allocated in that region. O3 states the encapsulation property of our system, that if y is inside the encapsulation boundary of z and x is outside, then x cannot access y.[1] (An object x *accesses* an object y if x has a pointer to y, or methods of x obtain a pointer to y.) Figure 8-6 shows an example ownership relation. We draw a solid line from x to y if x owns y. Region `r2` owns `s1`, `s1` owns `s1.head` and `s1.head.next`, etc.

[1]Our system handles inner class objects specially to support constructs like iterators. Details can be found in [44].

R1. For any region r, `heap` $\succeq r$ and `immortal` $\succeq r$.

R2. $x \succeq_o y \Longrightarrow x \succeq y$.

R3. If region $r_1 \succeq_o$ object o_1, region $r_2 \succeq_o$ object o_2, and $r_2 \not\succeq r_1$, then o_1 cannot contain a pointer to o_2.

Figure 8-2: Outlives Properties

Outlives Relation Our system allows programs to create regions. It also provides two special regions: the garbage collected region `heap`, and the "immortal" region `immortal`. The lifetime of a region is the time interval from when the region is created until it is deleted. If the lifetime of a region r_1 includes the lifetime of region r_2, we say that r_1 *outlives* r_2, and write $r_1 \succeq r_2$. The relation $\succeq$ is thus reflexive and transitive. We extend the *outlives* relation to include objects. We define that $x \succeq_o y$ implies $x \succeq y$. The extension is natural: if object o_1 owns object o_2 then o_1 outlives o_2 because o_2 is accessible only through o_1. Also, if region r owns object o then r outlives o because o is allocated in r. Our outlives relation has the properties shown in Figure 8-2. R1 states that `heap` and `immortal` outlive all regions. R2 states that the outlives relation includes the ownership relation. R3 states our memory safety property, that if object o_1 in region r_1 contains a pointer to object o_2 in region r_2, then r_2 outlives r_1. R3 implies that there are no dangling references in our system. Figure 8-6 shows an example outlives relation. We draw a dashed line from region x to region y if x outlives y. In the example, region `r1` outlives region `r2`, and `heap` and `immortal` outlive all regions. The following lemmas follow trivially from the above definitions:

Lemma 3 *If object $o_1 \succeq$ object o_2, then $o_1 \succeq_o o_2$.*

Lemma 4 *If region $r \succeq$ object o, then there exists a unique region r' such that $r \succeq r'$ and $r' \succeq_o o$.*

Grammar To simplify the presentation of key ideas behind our approach, we describe our type system formally in the context of a core subset of Java known as Classic Java [93]. Our approach, however, extends to the whole of Java and other similar languages. Figure 8-3 presents the grammar for our core language. A program consists of a series of class declarations followed by an initial expression. A predefined class `Object` is the root of the class hierarchy.

Owner Polymorphism Every class definition is parameterized with one or more owners. (This is similar to parametric polymorphism [3, 48, 153] except that our parameters are values, not types.) An owner can be an object or a region. Parameterization allows programmers to implement a generic class whose objects can have different owners. The first formal owner is special: it owns the corresponding object; the other owners propagate the ownership information. Methods can also declare an additional list of formal owner parameters. Each time new formals are introduced,

P	$::=$	$\mathit{def}^*\ e$
def	$::=$	$\texttt{class}\ cn\langle \mathit{formal}+\rangle\ \texttt{extends}\ c$ $\texttt{where}\ \mathit{constr}^*\ \{\ \mathit{field}^*\ \mathit{meth}^*\ \}$
formal	$::=$	$k\ fn$
c	$::=$	$cn\langle \mathit{owner}+\rangle \mid \texttt{Object}\langle \mathit{owner}\rangle$
owner	$::=$	$fn \mid r \mid \texttt{this} \mid \texttt{initialRegion} \mid \texttt{heap} \mid \texttt{immortal}$
field	$::=$	$t\ fd$
meth	$::=$	$t\ \ mn\langle \mathit{formal}^*\rangle\texttt{((}t\ \ p\texttt{)}^*\texttt{)}\ \texttt{where}\ \mathit{constr}^*\ \{\ e\ \}$
constr	$::=$	$\mathit{owner}\ \texttt{owns}\ \mathit{owner} \mid \mathit{owner}\ \texttt{outlives}\ \mathit{owner}$
t	$::=$	$c \mid \texttt{int} \mid \texttt{RHandle}\langle r\rangle$
k	$::=$	$\texttt{Owner} \mid \texttt{ObjOwner} \mid \mathit{rkind}$
rkind	$::=$	$\texttt{Region} \mid \texttt{GCRegion} \mid \texttt{NoGCRegion} \mid \texttt{LocalRegion}$
e	$::=$	$v \mid \texttt{let}\ v\ \texttt{=}\ e\ \texttt{in}\ \{\ e\ \} \mid v.fd \mid v.fd = v \mid \texttt{new}\ c \mid$ $v\texttt{.}mn\langle \mathit{owner}^*\rangle\texttt{(}v^*\texttt{)} \mid \texttt{(RHandle}\langle r\rangle\ h\texttt{)}\ \{\ e\ \}$
h	$::=$	v
cn	$\in$	class names
fd	$\in$	field names
mn	$\in$	method names
fn	$\in$	formal identifiers
v, p	$\in$	variable names
r	$\in$	region identifiers

Figure 8-3: Grammar for Object Oriented Programs

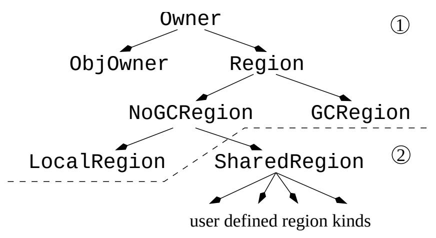


Figure 8-4: Owner Kind Hierarchy: Section 8.2.1 uses only Area 1. Sections 8.2.2 & 8.2.3 use Areas 1 & 2.

programmers can specify constraints between them using `where` clauses [73]. The constraints have the form "o_1 `owns` o_2" (i.e., $o_1 \succeq_o o_2$) and "o_1 `outlives` o_2" (i.e., $o_1 \succeq o_2$).

Each formal has an owner kind. There is a subkinding relation between owner kinds, resulting in the kind hierarchy from the upper half of Figure 8-4. The hierarchy is rooted in `Owner`, that has two subkinds: `ObjOwner` (owners that are objects; we avoid using `Object` because it is already used for the root of the class hierarchy) and `Region`. `Region` has two subkinds: `GCRegion` (the kind of the garbage collected heap) and `NoGCRegion` (the kind of other regions). Finally, `NoGCRegion` has a single subkind, `LocalRegion`. (At this point, there is no distinction between `NoGCRegion` and `LocalRegion`. We will add new kinds in the next section.)

Region Creation The expression "`(RHandle`$\langle r\rangle$ h`)` $\{e\}$" creates a new region and introduces two identifiers r and h that are visible inside the scope of e. r is an owner of kind `LocalRegion` that is bound to the newly created region. h is a runtime value of type `RHandle`$\langle r\rangle$ that is bound to the *handle* of the region r. The region name r is only a compile-time entity; it is erased (together with all the ownership and region type annotations) immediately after typechecking. However, the region handle h is required at runtime when we allocate objects in region r (object allocation is explained in the next paragraph). The newly created region is outlived by all regions that existed when it was created; it is destroyed at the end of the scope of e. This implies a "last in first out" order on region lifetimes. As we mentioned before, in addition to the user created regions, we have special regions: the garbage collected region `heap` (with handle `h_heap`) and the "immortal" region `immortal` (with handle `h_immortal`). Objects allocated in the `immortal` region are never deallocated. `heap` and `immortal` are never destroyed; hence, they outlive all regions. We also allow methods to allocate objects in the special region `initialRegion`, which denotes the most recent region that was created before the method was called. We use runtime support to acquire the handle of `initialRegion`.

Object Creation New objects are created using the expression "`new` $cn\langle o_{1..n}\rangle$". o_1 is the owner of the new object. (Recall that the first owner parameter always owns the corresponding object.) If o_1 is a region, the new object is allocated there; otherwise, it is allocated in the region where the object o_1 is allocated. For the purpose of typechecking, region handles are unnecessary. However, at runtime, we need the handle of the region we allocate in. The typechecker checks that we can obtain such a handle (more details are in Section 8.2.4). If o_1 is a region r, the handle of r must be in the environment. Therefore, if a method has to allocate memory in a specific region that is passed to it as an owner parameter, then it also needs to receive the corresponding region handle as an argument.

A formal owner parameter can be instantiated with an in-scope formal, a region name, or the `this` object. For every type $cn\langle o_{1..n}\rangle$ with multiple owners, our type system statically enforces the constraint that $o_i \succeq o_1$, for all $i \in \{1..n\}$. In addition, if an object of type $cn\langle o_{1..n}\rangle$ has a method mn, and if a formal owner parameter of mn is instantiated with an object obj, then our system ensures that $obj \succeq o_1$. These restrictions enable the type system to statically enforce object encapsulation and prevent dangling references.

Example We illustrate our type system with the example in Figure 8-5. A `TStack` is a stack of `T` objects. It is implemented using a linked list. The `TStack` class is parameterized by `stackOwner` and `TOwner`. `stackOwner` owns the `TStack` object and `TOwner` owns the `T` objects contained in the `TStack`. The code specifies that the `TStack` object owns the nodes in the list; therefore the list nodes cannot be accessed from outside the `TStack` object. The program creates two regions `r1` and `r2` such that `r1` outlives `r2`. The program declares several `TStack` variables: the type of `TStack s1` specifies that it is allocated in region `r2` and so are the `T` objects in `s1`; `TStack s2` is allocated in region `r2` but the `T` objects in `s2` are allocated in region `r1`; etc. Note that the type of `s6` is illegal. This is because `s6` is declared as `TStack`⟨`r1`,`r2`⟩, and `r2` $\not\succeq$ `r1`. (Recall that in any legal type $cn\langle o_{1..n}\rangle$ with multiple owners, $o_i \succeq o_1$ for all $i \in \{1..n\}$.) Figure 8-6 presents the ownership and the outlives relations from this example (assuming the stacks contain two elements each). We use circles for objects, rectangles for regions, solid arrows for ownership, and dashed arrows for the outlives relation between regions.

```
class TStack<Owner stackOwner, Owner TOwner> {
  TNode<this, TOwner> head = null;

  void push(T<TOwner> value) {
    TNode<this, TOwner> newNode = new TNode<this, TOwner>;
    newNode.init(value, head);  head = newNode;
  }

  T<TOwner> pop() {
    if(head == null) return null;
    T<TOwner> value = head.value;  head = head.next;
    return value;
  }
}

class TNode<Owner nodeOwner, Owner TOwner> {
  T<TOwner> value;
  TNode<nodeOwner, TOwner> next;

  void init(T<TOwner> v, TNode<nodeOwner, TOwner> n) {
    this.value = v;  this.next = n;
  }
}

(RHandle<r1> h1) {
  (RHandle<r2> h2) {
    TStack<r2,       r2>       s1;
    TStack<r2,       r1>       s2;
    TStack<r1,       immortal> s3;
    TStack<heap,     immortal> s4;
    TStack<immortal, heap>     s5;
 /* TStack<r1,       r2>       s6; illegal! */
 /* TStack<heap,     r1>       s7; illegal! */
}}
```

Figure 8-5: Stack of T Objects

Safety Guarantees The following two theorems state our safety guarantees. Part 1 of Theorems 5 and 6 state the object encapsulation property. Note that objects owned by regions are not encapsulated within other objects. Part 2 of Theorem 5 states the memory safety property.

Theorem 5 *If objects o_1 and o_2 are allocated in regions r_1 and r_2 respectively, and field fd of o_1 points to o_2, then*

1. *Either owner of $o_2 \succeq_o o_1$, or owner of o_2 is a region.*
2. *Region r_2 outlives region r_1.*

Proof: Suppose *class* $cn\langle f_{1..n}\rangle\{...\ T\langle x_1, ...\rangle\ fd\ ...\}$ is the class of o_1. Field *fd* of type $T\langle x_1, ...\rangle$ contains a reference to o_2. x_1 must therefore own o_2. x_1 can be either 1) `heap`, or 2) `immortal`, or 3) `this`, or 4) f_i, a class formal. In the first two cases, (owner of o_2) $= x_1$ is a region, and $r_2 = x_1 \succeq r_1$. In Case 3, (owner of o_2) $= o_1 \succeq_o o_1$, and $r_2 = r_1 \succeq r_1$. In Case 4, we know that $f_i \succeq f_1$, since all owners in a legal type outlive the first owner. Therefore, (owner of o_2) $= x_1 = f_i \succeq f_1 \succeq$ `this` $= o_1$. If (owner of o_2) is an object, we know from Lemma 3 that (owner of o_2) $\succeq_o o_1$. This also implies that $r_2 = r_1 \succeq r_1$. If the (owner of o_2) is a region, we know from Lemma 4 that there exists region r such that (owner of o_2) $\succeq r$ and $r \succeq_o o_1$. Therefore $r_2 = r \succeq r_1$.

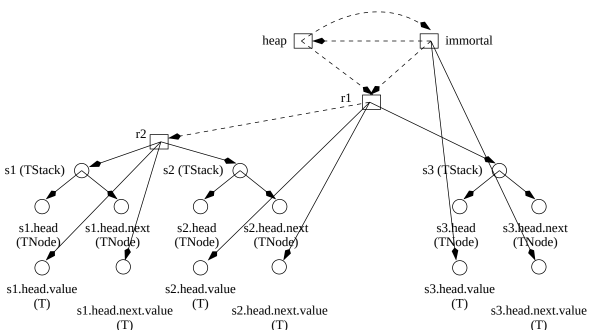


Figure 8-6: TStack Ownership and Outlives Relations

Theorem 6 *If a variable v in a method mn of an object o_1 points to an object o_2, then*

1. *Either owner of $o_2 \succeq_o o_1$, or owner of o_2 is a region.*

Proof: Similar to the proof of Theorem 5, except that now we have a fifth possibility for the (owner of o_2): a formal method parameter that is a region or `initialRegion` (that are not required to outlive o_1). In this case, (owner of o_2) is a region. The other four cases are identical.

Most previous region type systems allow programs to create, but not follow, dangling references. Such references can cause a safety problem when used with moving collectors. Our system therefore prevents a program from creating dangling references in the first place. Part 2 of Theorem 5 prevents object fields from containing dangling references. Even though Theorem 6 does not have a similar Part 2, we can prove, using lexical scoping of region names, that local variables cannot contain dangling references either.

8.2.2 Regions for Multithreaded Programs

This section describes how we support multithreaded programs. Figure 8-7 presents the language extensions. A `fork` instruction spawns a new thread that evaluates the invoked method. The evaluation is performed only for its effect; the parent thread does not wait for the completion of the new thread and does not use the result of the method call. Our unstructured concurrency model (similar to Java's model) is incompatible with the regions from Section 8.2.1 whose lifetimes are lexically bound. Those regions can still be used for allocating thread-local objects (hence the name of the associated region kind, `LocalRegion`), but objects shared by multiple threads require *shared* regions, of kind `SharedRegion`.

Shared Regions "(`RHandle`$\langle$*rkind* $r\rangle$ h) $\{e\}$" creates a shared region (*rkind* specifies the region kind of r; region kinds are explained later in this section). Inside expression e, the identifiers r and h are bound to the region and the region handle, respectively. Inside e, r and h can be passed to child threads. The objects allocated inside a shared region are not deleted as long as some thread can still access them. To ensure this, each thread maintains a stack of shared regions it can access, and each shared region maintains a counter of how many such stacks it is an element of. When

```
       P  ::=  def* srkdef* e
  srkdef  ::=  regionKind srkn⟨formal*⟩ extends srkind
                  where constr* { field* subsreg* }
   rkind  ::=  ... as in Figure 8-3 ... | srkind
  srkind  ::=  srkn⟨owner*⟩ | SharedRegion
 subsreg  ::=  srkind rsub
       e  ::=  ... as in Figure 8-3 ... |
               fork v.mn⟨owner*⟩(v*) |
               (RHandle⟨rkind r⟩ h) { e } |
               (RHandle⟨r⟩ h = [new]opt h.rsub) { e } |
               h.fd | h.fd = v

    srkn  ∈    shared region kind names
    rsub  ∈    shared subregion names
```

Figure 8-7: Extensions for Multithreaded Programs

a new shared region is created, it is pushed onto the region stack of the current thread and its counter is initialized to one. A child thread inherits all the shared regions of its parent thread; the counters of these regions are incremented when the child thread is forked. When the scope of a region name ends (the names of the shared regions are still lexically scoped, even if the lifetimes of the regions are not), the corresponding region is popped off the stack and its counter is decremented. When a thread terminates, the counters of all the regions from its stack are decremented. When the counter of a region becomes zero, the region is deleted. The typing rule for a `fork` expression checks that objects allocated in local regions are not passed to the child thread as arguments; it also checks that local regions and handles to local regions are not passed to the child thread.

Subregions and Portals Shared regions provide the basis for inter-thread communication. However, in many cases, they are not enough. E.g., consider two long-lived threads, a producer and a consumer, that communicate through a shared region in a repetitive way. In each iteration, the producer allocates some objects in the shared region and the consumer subsequently uses the objects. These objects become unreachable after each iteration. However, these objects are not deleted until both threads terminate and exit the shared region. To prevent this memory leak, we allow shared regions to have subregions. In each iteration, the producer and the consumer can enter a subregion of the shared region and use it for communication. At the end of the iteration, both the threads exit the subregion and the reference count of the subregion goes to zero—the objects in the subregion are thus deleted after each iteration.

We must also allow the producer to pass references to objects it allocates in the subregion in each iteration to the consumer. Note that storing the references in the fields of a "hook" object is not possible: objects allocated outside the subregion cannot point to objects in the subregion (otherwise, those references would result in dangling references when objects in the subregion are deleted), and objects allocated in the subregion do not survive between iterations and hence cannot be used as "hooks". To solve this problem, we allow (sub)regions to contain *portal* fields. A thread can store the reference to an object in a portal field; other threads can then read the portal field to obtain the reference.

```
regionKind BufferRegion extends SharedRegion {
  BufferSubRegion b;
}

regionKind BufferSubRegion extends SharedRegion {
  Frame<this> f;
}

class Producer<BufferRegion r> {
  void run(RHandle<r> h) {
    while(true) {
      (RHandle<BufferSubRegion r2> h2 = h.b) {
        Frame<r2> frame = new Frame<r2>;
        get_image(frame);
        h2.f = frame;
      }
      ... // wake up the consumer
      ... // wait for the consumer
}}}

class Consumer<BufferRegion r> {
  void run(RHandle<r> h) {
    while(true) {
      ... // wait for the producer
      (RHandle<BufferSubRegion r2> h2 = h.b) {
        Frame<r2> frame = h2.f;
        h2.f = null;
        process_image(frame);
      }
      ... // wake up the producer
}}}

(RHandle<BufferRegion r> h) {
  fork (new Producer<r>).run(h);
  fork (new Consumer<r>).run(h);
}
```

Figure 8-8: Producer Consumer Example

Region Kinds In practice, programs can declare several shared region kinds. Each such kind extends another shared region kind and can declare several portal fields and subregions (see grammar rule for *srkdef* in Figure 8-7). The resulting shared region kind hierarchy has `SharedRegion` as its root. The owner kind hierarchy now includes both Areas 1 and 2 from Figure 8-4. Similar to classes, shared region kinds can be parameterized with owners; however, unlike objects, regions do not have owners so there is no special meaning attached to the first owner.

Expression "(`RHandle`$\langle r_2 \rangle$ h_2 = [`new`]$_{\text{opt}}$ h_1.*rsub*) {e}" evaluates e in an environment where r_2 is bound to the subregion *rsub* of the region r_1 that h_1 is the handle of, and h_2 is bound to the handle of r_2. In addition, if the keyword `new` is present, r_2 is a newly created subregion, distinct from the previous *rsub* subregion.

If h is the handle of region r, the expression "h.*fd*" reads r's portal field *fd*, and "h.*fd* = v" stores a value into that field. The rule for portal fields is the same as that for object fields: a portal field of a region r is either null or points to an object allocated in r or in a region that outlives r.

Flushing Subregions When all the objects in a subregion become inaccessible, the subregion is flushed, i.e., all objects allocated inside it are deleted. We do not flush a subregion if its counter is positive. Furthermore, we do not flush a subregion r if any of its portal fields is non-null (to allow some thread to enter it later and use those objects) or if any of r's subregions has not been flushed yet (because the objects in those subregions might point to objects in r). Recall that subregions are a way of "packaging" some data and sending it to another thread; the receiver thread looks inside the subregion (starting from the portal fields) and uses the data. Therefore, as long as a subregion with non-null portal fields is reachable (i.e., a thread may obtain its handle), the objects allocated inside it can be reachable even if no thread is currently in the subregion.

Example Figure 8-8 contains an example that illustrates the use of subregions and portal fields. The main thread creates a shared region of kind `BufferRegion` and then starts two threads, a producer and a consumer, that communicate through the shared region. In each iteration, the producer enters subregion `b` (of kind `BufferSubRegion`), allocates a `Frame` object in it, and stores a reference to the frame in subregion's portal field `f`. Next, the producer exits the subregion and waits for the consumer. The subregion is not flushed because the portal field `f` is non-null. The consumer then enters the subregion, uses the frame object pointed to by its portal field `f`, sets `f` to `null`, and exits the subregion. Now, the subregion is flushed (because its counter is zero and all its fields are null) and a new iteration starts. In this chapter, we do not discuss synchronization issues; we assume synchronization primitives similar to those in Java.

8.2.3 Regions for Real-Time Programs

A real-time program consists of a set of real-time threads, a set of regular threads, and a special garbage collector thread. (This is a conceptual model; actual implementations might differ.) A real-time thread has strict deadlines for completing its tasks.[2]

Figure 8-9 presents the language extensions to support real-time programs. The expression "`RT_fork` $v.mn\langle owner^*\rangle(v^*)$" spawns a new real-time thread to evaluate mn. Such a thread cannot afford to be interrupted for an unbounded amount of time by the garbage collector—the rest of this section explains how our type system statically ensures this property.

Effects The garbage collector thread must synchronize with any thread that creates or destroys heap roots, i.e., references to heap objects, otherwise it might end up collecting reachable objects. Therefore, we must ensure that the real-time threads do not read or overwrite references to heap objects. (The last restriction is needed to support moving collectors.) To statically check this, we allow methods to declare *effects* clauses [149]. In our system, the effects clause of a method lists the owners (some of them regions) that the method *accesses*. Accessing a region means allocating an object in that region. Accessing an object means reading or overwriting a reference

[2]Our terminology is related, but not identical to the RTSJ terminology. E.g., our real-time threads are similar to (and more restrictive than) the RTSJ `NoHeapRealtimeThreads`.

```
    meth  ::=  t mn⟨formal*⟩((t p)*) effects where constr* {e}
 effects  ::=  accesses owner*
   owner  ::=  ... as in Figure 8-3 ... | RT
 subsreg  ::=  srkind : rpol tt rsub
    rpol  ::=  LT(size) | VT
      tt  ::=  RT | NoRT
       k  ::=  ... as in Figure 8-3 ... | rkind : LT
       e  ::=  ... as in Figure 8-7 ... |
               (RHandle⟨rkind : rpol r⟩ h) { e } |
               RT_fork v.mn⟨owner*⟩(v*)
```

Figure 8-9: Extensions for Real-Time Programs

to that object or allocating another object owned by that object. Note that we do not consider reading or writing a field of an object as accessing that object. If a method's effects clause consists of owners $o_{1..n}$, then any object or region accessed by that method, the methods it invokes, and the threads it spawns (transitively) is guaranteed to be outlived by o_i, for some $i \in \{1..n\}$.

The typing rule for an `RT_fork` expression checks all the constraints of a regular `fork` expression. In addition, it checks that references to heap objects are not passed as arguments to the new thread, and that the effects clause of the method evaluated in the new thread does not contain the heap region or any object allocated in the heap region. If an `RT_fork` expression typechecks, the new real-time cannot receive any heap reference. Furthermore, it cannot create a heap object, or read or overwrite a heap reference in an object field—the type system ensures that in each of the above cases, the heap region or an object allocated in the heap region appears in the method effects.

Region Allocation Policies A real-time thread cannot create an object if this operation requires allocating new memory, because allocating memory requires synchronization with the garbage collector. A real-time thread can, however, create an object in a preallocated memory region.

Our system supports two allocation policies for regions. One policy is to allocate memory on demand (potentially in large chunks), as new objects are created in the region. Allocating a new object can take unbounded time or might not even succeed (if a new chunk is needed and the system runs out of memory). Flushing the region frees all the memory allocated for that region. Following the RTSJ terminology, we call such regions *VT* (Variable Time) regions.

The other policy is to allocate all the memory for a region at region creation time. The programmer must provide an upper bound for the total size of the objects that will be allocated in the region. Allocating an object requires sliding a pointer—if the region is already full, the system throws an exception to signal that the region size was too small. Allocating a new object takes time linear in its size: sliding the pointer takes constant time, but we also have to set to zero each allocated byte. Flushing the region simply resets a pointer, and, importantly, *does not* free the memory allocated for the region. We call regions that use this allocation policy *LT* (Linear Time) regions. Once we have an LT subregion, threads can repeatedly enter it, allocate objects in it, exit it (thus flushing it), and re-enter it without having to allocate new memory. This is possible because flushing an LT region does not free its memory.

LT subregions are thus ideal for real-time threads: once such a subregion is created (with a large enough upper bound), all object creations will succeed, in linear time; moreover, the subregion can be flushed and reused without memory allocation.

We allow users to specify the region allocation policy (LT or VT) when a new region is created. The policy for subregions is declared in the shared region kind declarations. When a user specifies an LT policy, the user also has to specify the size of the region (in bytes). An expression "`(RHandle`$\langle$*rkind* : *rpol r*$\rangle$ *h*`)` {*e*}" creates a region with allocation policy *rpol* and allocates memory for all its (transitive) LT (sub)regions (including itself). Our system checks that a region has a finite number of transitive subregions.

If a method enters a VT region or a top level region (i.e., a region that is not a subregion), the typechecker ensures that the method contains the heap region in its effects clause. This is to prevent real-time threads from invoking such methods. However, a method that does not contain the heap region in its effects clause can still enter an existing LT subregion, because no memory is allocated in that case.

Preventing the RTSJ Priority Inversion So far, we presented techniques for checking that real-time threads do not create or destroy heap references, create new regions, or allocate objects in VT regions. However, there are two other subtle ways a thread can interact with the garbage collector.

First, the garbage collector needs to know all locations that refer to heap objects, including locations that are inside regions. Suppose a real-time thread uses an LT region that contains such heap references (created by a non-real-time thread). The real-time thread can flush the region (by exiting it) thus destroying any heap reference that existed in the region. If we use a moving garbage collector, the real-time thread has to interact with the garbage collector to inform it about the destruction of those heap references. Therefore, we should prevent regions that can be flushed by a real-time thread from containing any heap reference (even if the reference is not explicitly read or overwritten by the real-time thread). Note that this restriction is relevant only for subregions: a real-time thread cannot create a top-level region and hence cannot flush a top-level region either.

Second, when a thread enters or exits a subregion, it needs to do some bookkeeping. To preserve the integrity of the runtime region implementation, some synchronization is necessary during this bookkeeping. E.g., when a thread exits a subregion, the test that the subregion can be flushed and the actual flushing have to be executed atomically, without allowing any thread to enter the subregion "in between". If a regular thread exiting a subregion is suspended by the garbage collector, then a real-time thread entering the subregion might have to wait for an unbounded amount of time. This *priority inversion* problem occurs even in the RTSJ.

To prevent these subtle interactions, we impose the restriction that real-time threads and regular threads cannot share subregions. Subregions used by real-time threads thus cannot contain heap references, and real-time threads never have to wait for unbounded amounts of time.

For each subregion, programmers specify in the region kind definitions whether the subregion will be used only by real-time threads (`RT` subregions) or only by regular threads (`NoRT` subregions). Note that real-time and regular threads can still communicate using top-level regions. Any method that enters an `RT` subregion must contain the special effect `RT` in its effects clause. Any method that enters a `NoRT` subregion must contain the heap region in its effects clause. The type system checks that no regular thread can invoke a method that has an `RT` effect, and no real-time thread can invoke a method that has a heap effect.

8.2.4 Rules for Typechecking

Previous sections presented the grammar for our core language in Figures 8-3, 8-7, and 8-9. This section presents some sample typing rules; [47] contains all the rules.

The core of our type system is a set of typing judgments of the form $P; E; X; r_{cr} \vdash e : t$. P, the program being checked, is included to provide information about class definitions. The typing environment E provides information about the type of the free variables of e ($t\ v$, i.e., variable v has type t), the kind of the owners currently in scope ($k\ o$, i.e., owner o has kind k), and the two relations between owners: the "ownership" relation ($o_2 \succeq_o o_1$, i.e., o_2 `owns` o_1) and the "outlives" relation ($o_2 \succeq o_1$, i.e., o_2 `outlives` o_1). More formally, $E ::= \emptyset \mid E,\ t\ v \mid E,\ k\ o \mid E,\ o_2 \succeq_o o_1 \mid E,\ o_2 \succeq o_1$. r_{cr} is the current region. X must subsume the effects of e. t is the type of the expression e.

A useful auxiliary rule is $E \vdash X_1 \succeq X_2$, i.e., the effects X_1 *subsume* the effects X_2: $\forall o \in X_2, \exists g \in X_1$, s.t. $g \succeq o$. To prove constraints of the form $g \succeq o$, $g \succeq_o o$ etc. in a specific environment E, the checker uses the constraints from E, and the properties of $\succeq$ and $\succeq_o$: transitivity, reflexivity, $\succeq_o$ implies $\succeq$, and the fact that the first owner from the type of an object owns the object.

The expression "$\texttt{(RHandle}\langle r\rangle\ h\texttt{)}\ \{e\}$" creates a local region and evaluates e in an environment where r and h are bound to the new region and its handle respectively. The associated typing rule is presented below:

[EXPR LOCAL REGION]

$$\frac{\begin{array}{c} E_2 = E,\ \texttt{LocalRegion}\ r,\ \texttt{RHandle}\langle r\rangle\ h,\ (r_e \succeq r)_{\forall r_e \in Regions(E)} \\ P \vdash_{\text{env}} E_2 \quad P; E_2; X, r; r \vdash e : t \quad E \vdash X \succeq \texttt{heap} \end{array}}{P; E; X; r_{cr} \vdash \texttt{(RHandle}\langle r\rangle\ h\texttt{)}\ \{e\} : \texttt{int}}$$

The rule starts by constructing an environment E_2 that extends the original environment E by recording that r has kind `LocalRegion` and h has type $\texttt{RHandle}\langle r\rangle$. As r is deleted at the end of e, all existing regions outlive it; E_2 records this too ($Regions(E)$ denotes the set of all regions from E). e should typecheck in the context of the environment E_2 and the permitted effects are X, r (the local region r is a permitted effect inside e). Because creating a region requires memory allocation, X must subsume `heap`. The expression is evaluated only for its side-effects and its result is never used. Hence, the type of the entire expression is `int`.

The rule for a field read expression "$v.fd$" first finds the type $cn\langle o_{1..n}\rangle$ for v. Next, it verifies that fd is a field of class cn; let t be its declared type. The rule obtains the type of the entire expression by substituting in t each formal owner parameter fn_i of cn with the corresponding owner o_i:

[EXPR REF READ]

$$\frac{\begin{array}{c} P; E; X; r_{cr} \vdash v : cn\langle o_{1..n}\rangle \quad P \vdash (t\ fd) \in cn\langle fn_{1..n}\rangle \\ t' = t[o_1/fn_1]..[o_n/fn_n] \\ ((t' = \texttt{int}) \ \vee \ (t' = cn'\langle o'_{1..m}\rangle \ \wedge \ E \vdash X \succeq o'_1)) \end{array}}{P; E; X; r_{cr} \vdash v.fd : t'}$$

The last line of the rule checks that if the expression reads an object reference (i.e., not an integer), then the list of effects X subsumes the owner of the referenced object.

For an object allocation expression "$\texttt{new}\ \ cn\langle o_{1..n}\rangle$", the rule first checks that class cn is defined in P:

[EXPR NEW]

$$\frac{\begin{array}{c} \texttt{class}\ \ cn\langle(k_i\ \ fn_i)_{i\in\{1..n\}}\rangle\ \ ...\ \ \texttt{where}\ \ constr_{1..c}\ \ ... \in P \\ \forall i = 1..m,\ (E \vdash_{\mathrm{k}} o_i : k'_i\ \ \wedge\ \ P \vdash k'_i \leq_{\mathrm{k}} k_i\ \ \wedge\ \ E \vdash o_i \succeq o_1) \\ \forall i = 1..c,\ E \vdash\ constr_i[o_1/fn_1]..[o_m/fn_m] \\ E \vdash X \succeq o_1 \quad E \vdash_{\mathrm{av}} \texttt{RH}(o_1) \end{array}}{P; E; X; r_{cr} \vdash \texttt{new}\ \ cn\langle o_{1..n}\rangle : cn\langle o_{1..n}\rangle}$$

Next, it checks that each formal owner parameter fn_i of cn is instantiated with an owner o_i of appropriate kind, i.e., the kind k'_i of o_i is a subkind of the declared kind k_i of fn_i. It also checks that in E, each owner o_i outlives the first owner o_1, and each constraint of cn is satisfied. Allocating an object means accessing its owner; therefore, X must subsume o_1. The new object is allocated in the region o_1 (if o_1 is a region) or in the region that o_1 is allocated in (if o_1 is an object). The last part of the precondition, $E \vdash_{\mathrm{av}} \texttt{RH}(o_1)$, checks that the handle for this region is available. To prove facts of this kind, the type system uses the following rules:

[AV HANDLE]

$$\frac{E = E_1,\ \texttt{RHandle}\langle r\rangle\ h,\ E_2}{E \vdash_{\mathrm{av}} \texttt{RH}(r)}$$

[AV THIS]

$$\frac{}{E \vdash_{\mathrm{av}} \texttt{RH(this)}}$$

[AV TRANS1]

$$\frac{E \vdash o_1 \succeq_o o_2 \quad E \vdash_{\mathrm{av}} \texttt{RH}(o_2)}{E \vdash_{\mathrm{av}} \texttt{RH}(o_1)}$$

[AV TRANS2]

$$\frac{E \vdash o_1 \succeq_o o_2 \quad E \vdash_{\mathrm{av}} \texttt{RH}(o_1)}{E \vdash_{\mathrm{av}} \texttt{RH}(o_2)}$$

The rule [AV HANDLE] looks for a region handle in the environment. The environment always contains handles for `heap` and `immortal`; in addition, it contains all handle identifiers that are in scope. The rule [AV THIS] reflects the fact that our runtime is able to find the handle of the region where an object (`this` in particular) is allocated. The last two rules use the fact that all objects are allocated in the same region as their owner. Therefore, if $o_1 \succeq_o o_2$ and the region handle for one of them is available, then the region handle for the other one is also available. Note that these rules do significant reasoning, thus reducing annotation burden; e.g., if a method allocates only objects (transitively) owned by `this`, it does not need an explicit region handle argument.

We end this section with the typing rule for `fork`. The rule first checks that the method call is well-typed (see rule [EXPR INVOKE] in [47]) Note that *mn* cannot have the `RT` effect: a non-real-time thread cannot enter a subregion that is reserved only for real-time threads.

[EXPR FORK]

$$\frac{\begin{array}{c} P; E; X \setminus \{\texttt{RT}\}; r_{cr} \vdash v_0 \,.\, mn\langle o_{(n+1)..m}\rangle(v_{1..u}) : t \\ \mathit{NonLocal}(k) \overset{def}{=} (P \vdash k \leq_{\mathrm{k}} \texttt{SharedRegion}) \;\vee\; (P \vdash k \leq_{\mathrm{k}} \texttt{GCRegion}) \\ E \vdash \mathrm{RKind}(r_{cr}) = k_{cr} \quad \mathit{NonLocal}(k_{cr}) \\ P; E; X; r_{cr} \vdash v_0 : cn\langle o_{1..n}\rangle \\ \forall i = 1..m,\ (E \vdash \mathrm{RKind}(o_i) = k_i \;\wedge\; \mathit{NonLocal}(k_i)) \end{array}}{P; E; X; r_{cr} \vdash \texttt{fork}\ v_0 \,.\, mn\langle o_{(n+1)..m}\rangle(v_{1..u}) : \texttt{int}}$$

The rule checks that the new thread does not receive any local region or objects allocated in a local region. It uses the following observation: the only owners that appear in the types of the method arguments are: `initialRegion`, `this`, the formals for the method and the formals for the class the method belongs to. Therefore, the arguments passed to the method *mn* from the `fork` instruction may be owned only by the current region at the point of the `fork`, by the owners $o_{1..n}$ that appear in the type of the object v_0 points to, or by the owners $o_{(n+1)..m}$ that appear explicitly in the `fork` instruction. For each such owner o, our system uses the rule $E \vdash \mathrm{RKind}(o) = k$ to extract the kind k of the region it stands for (if it is a region), or of the region it is allocated in (if it is an object). The rule next checks that k is a subkind of `SharedRegion` or `GCRegion`. The rules for inferring statements of the form $E \vdash \mathrm{RKind}(o_i) = k_i$ (see [47]) are similar to the previously explained rules for checking that a region handle is available. The key idea they exploit is that a subobject is allocated in the same region as its owner.

8.2.5 Type Inference

Although our type system is explicitly typed in principle, it would be onerous to fully annotate every method with the extra type information that our system requires. Instead, we use a combination of type inference and well-chosen defaults to significantly reduce the number of annotations needed in practice. Our system also supports user-defined defaults to cover specific patterns that might occur in user code. We emphasize that our approach to inference is purely intra-procedural and we do not infer method signatures or types of instance variables. Rather, we use a default completion of partial type specifications in those cases. This approach permits separate compilation.

The following are some defaults currently provided by our system. If owners of method local variables are not specified, we use a simple unification-based approach to infer the owners. The approach is similar to the ones in [46, 43]. For parameters unconstrained after unification, we use `initialRegion`. For unspecified owners in method signatures, we use `initialRegion` as the default. For unspecified owners in instance variables, we use the owner of `this` as the default. For static fields, we use `immortal` as the default. Our default `accesses` clauses contain all class and method owner parameters and `initialRegion`.

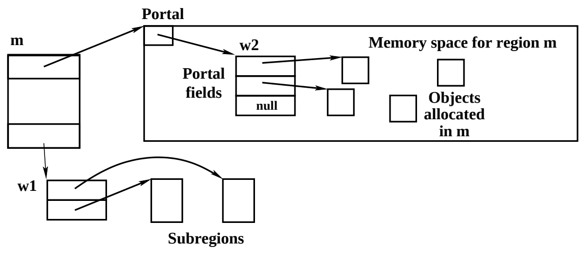


Figure 8-10: Translation of a Region With Three Fields and Two Subregions.

8.2.6 Translation to Real-Time Java

Although our system provides significant improvements over the RTSJ, programs in our language can be translated to RTSJ reasonably easily, by local translation rules. This is mainly because we designed our system so that it can be implemented using type erasure (region handles exist specifically for this purpose). Also, RTSJ has mechanisms that are powerful enough to support our features. RTSJ offers `LTMemory` and `VTMemory` regions where it takes linear time and variable time (respectively) to allocate objects. RTSJ regions are Java objects that point to some memory space. In addition, RTSJ has two special regions: `heap` and `immortal`. A thread can allocate in the current region using `new`. A thread can also allocate in any region that it entered using `newInstance`, which requires the corresponding region object. RTSJ regions are maintained similarly to our shared regions, by counting the number of threads executing in them. RTSJ regions have one *portal*, which is similar to a portal field except that its declared type is `Object`. Most of the translation effort is focused on providing the missing features: subregions and multiple, typed portal fields. We discuss the translation of several important features from our type system; the full translation is discussed in [190].

We represent a region r from our system as an RTSJ region `m` plus two auxiliary objects `w1` and `w2` (see Figure 8-10). `m` points to a memory area that is pre-allocated for an `LT` region, or grown on-demand for a `VT` region. `m` also points to an object `w1` whose fields point to the representation of r's subregions. (We subclass `LT/VTMemory` to add an extra field.) In addition, `m`'s portal points to an object `w2` that serves as a wrapper for r's portal fields. `w2` is allocated in the memory space attached to `m`, while `m` and `w1` are allocated in the region that was current at the time `m` was created.

The translation of "`new` $cn\langle o_{1..n}\rangle$" requires a reference to (i.e., the handle of) the region we allocate in. If this is the same as the current region, we use the more efficient `new`. The type rules already proved that we can obtain the necessary handle, i.e., $E \vdash_{\text{av}} \texttt{RH}(o_1)$; we presented the relevant type rules in Section 8.2.4. Those rules "pushed" the judgment $E \vdash_{\text{av}} \texttt{RH}(o)$ up and down the ownership relation until we obtained an owner whose region handle was available: `immortal`, `heap`, `this`, or a region whose region handle was available in a local variable. RTSJ provides mechanisms for retrieving the handle in the first three cases: `ImmortalArea.instance()`, `HeapArea.instance()`, and `MethodArea.getMethodArea(Object)`, respectively. In the last case, we simply use the handle from the local variable.

Program	Lines of Code	Lines Changed
Array	56	4
Tree	83	8
Water	1850	31
Barnes	1850	16
ImageRec	567	8
http	603	20
game	97	10
phone	244	24

Figure 8-11: Programming Overhead

Program	Execution Time (sec)		Overhead
	Static Checks	Dynamic Checks	
Array	2.24	16.2	7.23
Tree	4.78	23.1	4.83
Water	2.06	2.55	1.24
Barnes	19.1	21.6	1.13
ImageRec	6.70	8.10	1.21
load	0.667	0.831	1.25
cross	0.014	0.014	1.0
threshold	0.001	0.001	1
hysteresis	0.005	0.006	1
thinning	0.023	0.026	1.1
save	0.617	0.731	1.18

Figure 8-12: Dynamic Checking Overhead

8.3 Experience

To gain preliminary experience, we implemented several programs in our system. These include two micro benchmarks (Array and Tree), two scientific computations (Water and Barnes), several components of an image recognition pipeline (load, cross, threshold, hysteresis, and thinning), and several simple servers (http, game, and phone, a database-backed information sever). In our implementations, the primary data structures are allocated in regions (i.e., not in the garbage collected heap). In each case, once we understood how the program worked and decided on the memory management policy to use, adding the extra type annotations was fairly straightforward. Figure 8-11 presents a measure of the programming overhead involved. It shows the number of lines of code that needed type annotations. In most cases, we only had to change code where regions were created.

We also used our RTSJ implementation to measure the execution times of these programs both with and without the dynamic checks specified in the Real-Time Specification for Java. Figure 8-12 presents the running times of the benchmarks both with and without dynamic checks. Note that there is no garbage collection overhead in any of these running times because the garbage collector never executes. Our micro benchmarks (Array and Tree) were written specifically to maximize the checking overhead—our development goal was to maximize the ratio of assignments to other computation. These programs exhibit the largest performance increases—they run approximately 7.2 and 4.8 times faster, respectively, without checks. The performance improvements for the scientific programs and image processing components provide a more realistic picture of the dynamic checking overhead. These programs have more modest performance improvements, running up to 1.25 times faster without the checks. For the servers, the running time is dominated by the network processing overhead and check removal has virtually no effect. This chapter presents the overhead of dynamic referencing and assignment checks. For a detailed analysis of the performance of a full range of RTSJ features, see [67, 68].

8.4 Related Work

The seminal work in [196, 195] introduces a static type system for region-based memory management for ML. Our system extends this to object-oriented programs by combining the benefits of region types and ownership types in a unified type system framework. Our system extends region types to multithreaded programs by allowing long-lived threads to share objects without using the heap and without having memory leaks. Our system extends region types to real-time programs by ensuring that real-time threads do not interfere with the garbage collector.

One disadvantage with most region-based management systems is that they enforce a lexical nesting on region lifetimes; so objects allocated in a given region may become inaccessible long before the region is deleted. [9] presents an analysis that enables some regions to be deleted early, as soon as all of the objects in the region are unreachable. Other approaches include the use of linear types to control when regions are deleted [71, 74]. None of these approaches currently support object-oriented programs and the consequent subtyping, multithreaded programs with shared regions, or real-time programs with real-time threads (although it should be possible to extend them to do so). Conversely, it should also be possible to apply these techniques to our system. In fact, existing systems already combine ownership-based type systems and unique pointers [64, 46, 13].

RegJava [59] has a region type system for object-oriented programs that supports subtyping and method overriding. Cyclone [111] is a dialect of C with a region type system. Our work improves on these two systems by combining the benefits of ownership types and region types in a unified framework. An extension to Cyclone handles multithreaded programs and provides shared regions [110]. Our work improves on this by providing subregions in shared regions and portal fields in subregions, so that long-lived threads can share objects without using the heap and without having memory leaks. Other systems for regions [99, 100] use runtime checks to ensure memory safety. These systems are more flexible, but they do not statically ensure safety.

To our knowledge, ours is the first static type system for memory management in real-time programs. [76, 77] automatically translates Java code into RTSJ code using off-line dynamic analysis to determine the lifetime of an object. Unlike our system, this system does not require type annotations. It does, however, impose a runtime overhead and it is not safe because the dynamic analysis might miss some execution paths. Programmers can use this dynamic analysis to obtain suggestions for region type annotations. We previously used escape analysis [189] to remove RTSJ runtime checks [191]. However, the analysis is effective only for programs in which no object escapes the computation that allocated it. Our type system is more flexible: we allow a computation to allocate objects in regions that may outlive the computation.

Real-time garbage collection [27, 23] provides an alternative to region-based memory management for real-time programs. It has the advantage that programmers do not have to explicitly deal with memory management. The basic idea is to perform a fixed amount of garbage collection activity for a given amount of allocation. With fixed-size allocation blocks and in the absence of cycles, reference counting can deliver a real-time garbage collector that imposes no space overhead as compared with manual memory management. Copying and mark and sweep collectors, on the other hand, pay space to get bounded-time allocation. The amount of extra space depends on the maximum live heap size, the maximum allocation rate, and other memory management parameters. The additional space allows the collector to successfully perform allocations while it processes the heap to reclaim memory. To obtain the real-time allocation guarantee, the programmer must calculate the required memory management parameters, then use those values to provide the collector with the required amount of extra space. In contrast, region-based memory management provides an explicit mechanism that programmers can use to structure code based on their understanding of the memory usage behavior of a program; this mechanism may enable programmers to obtain a smaller space overhead. The additional development burden consists of grouping objects into regions and determining the maximum size of LT regions [103, 104].

8.5 Conclusions

The Real-Time Specification for Java (RTSJ) allows programs to create real-time threads and use region-based memory management. The RTSJ uses runtime checks to ensure memory safety. This chapter presents a static type system that guarantees that these runtime checks will never fail for well-typed programs. Our type system therefore 1) provides an important safety guarantee and 2) makes it possible to eliminate the runtime checks and their associated overhead. Our system also makes several contributions over previous work on region types. For object-oriented programs, it combines the benefits of region types and ownership types in a unified type system framework. For multithreaded programs, it allows long-lived threads to share objects without using the heap and without having memory leaks. For real-time programs, it ensures that real-time threads do not interfere with the garbage collector. Our experience indicates that our type system is sufficiently expressive and requires little programming overhead, and that eliminating the RTSJ runtime checks using a static type system can significantly decrease the execution time of real-time programs.

THIS PAGE WAS INTENTIONALLY LEFT BLANK

Chapter 9

Incrementalized Pointer and Escape Analysis

9.1 Introduction

Program analysis research has focused on two kinds of analyses: *local* analyses, which analyze a single procedure, and *whole-program* analyses, which analyze the entire program. Local analyses fail to exploit information available across procedure boundaries; whole-program analyses are potentially quite expensive for large programs and are problematic when parts of the program are not available in analyzable form.

This paper describes our experience *incrementalizing* an existing whole-program analysis so that it can analyze arbitrary regions of complete or incomplete programs. The new analysis can 1) analyze each method independently of its caller methods, 2) skip the analysis of potentially invoked methods, and 3) incrementally incorporate analysis results from previously skipped methods into an existing analysis result. These features promote a structure in which the algorithm executes under the direction of an analysis policy. The policy continuously monitors the analysis results to direct the incremental investment of analysis resources to those parts of the program that offer the most attractive return (in terms of optimization opportunities) on the invested resources. Our experimental results indicate that this approach usually delivers almost all of the benefit of the whole-program analysis, but at a fraction of the cost.

9.1.1 Analysis Overview

Our analysis incrementalizes an existing whole-program analysis for extracting points-to and escape information [202]. The basic abstraction in this analysis is a points-to escape graph. The nodes of the graph represent objects; the edges represent references between objects. In addition to points-to information, the analysis records how objects escape the currently analyzed region of the program to be accessed by unanalyzed regions. An object may escape to an unanalyzed caller via a parameter passed into the analyzed region or via the return value. It may also escape to a potentially invoked but unanalyzed method via a parameter passed into that method.

Finally, it may escape via a global variable or parallel thread. If an object does not escape, it is captured.

The analysis is flow sensitive, context sensitive, and compositional. Guided by the analysis policy, it performs an incremental analysis of the neighborhood of the program surrounding selected object allocation sites. When it first analyzes a method, it skips the analysis of all potentially invoked methods, but maintains enough information to reconstruct the result of analyzing these methods should it become desirable to do so. The analysis policy then examines the graph to find objects that escape, directing the incremental integration of (possibly cached) analysis results from potential callers (if the object escapes to the caller) or potentially invoked methods (if the object escapes into these methods). Because the analysis has complete information about captured objects, the goal is to analyze just enough of the program to capture objects of interest.

9.1.2 Analysis Policy

We formulate the analysis policy as a solution to an investment problem. At each step of the analysis, the policy can invest analysis resources in any one of several allocation sites in an attempt to capture the objects allocated at that site. To invest its resources wisely, the policy uses empirical data from previous analyses, the current analysis result for each site, and profiling data from a previous training run to estimate the marginal return on invested analysis resources for each site.

During the analysis, the allocation sites compete for resources. At each step, the policy invests its next unit of analysis resources in the allocation site that offers the best marginal return. When the unit expires, the policy recomputes the estimated returns and again invests in the (potentially different) allocation site with the best estimated marginal return. As the analysis proceeds and the policy obtains more information about each allocation site, the marginal return estimates become more accurate and the quality of the investment decisions improves.

9.1.3 Analysis Uses

We use the analysis results to enable a stack allocation optimization. If the analysis captures an object in its allocating method, the object is unreachable once the method returns. In this case, the generated code allocates the object in the activation record of its allocating method. If the object escapes the allocating method, but is captured in one or more of the methods that directly invoke the allocating method, the compiler inlines the allocating method into the capturing callers, then generates code to allocate the captured objects in the activation record of the caller. The success of this optimization depends on the characteristics of the application. The vast majority of the objects in our benchmark applications are allocated at a small subset of the allocation sites. For some applications the analysis is able to capture and stack allocate all of the objects allocated at these sites. In other applications these objects escape and the analysis finds few relevant optimization opportunities.

Other optimization uses include synchronization elimination, the elimination of `ScopedMemory` checks in Real-Time Java [38], and a range of traditional compiler optimizations. Potential software engineering uses include the evaluation of programmer hypotheses regarding points-to and escape information for specific objects, the discovery of methods with no externally visible side effects, and the extraction of information about how methods access data from the enclosing environment.

Because the analysis is designed to be driven by an analysis policy to explore only those regions of the program that are relevant to a specific analysis goal, we expect the analysis to be particularly useful in settings (such as dynamic compilers and interactive software engineering tools) in which it must quickly answer queries about specific objects.

9.1.4 Context

In general, a base analysis must have several key properties to be a good candidate for incrementalization: it must be able to analyze methods independently of their callers, it must be able to skip the analysis of invoked methods, and it must be able to recognize when a partial analysis of the program has given it enough information to apply the desired optimization. Algorithms that incorporate escape information are good candidates for incrementalization because they enable the analysis to recognize captured objects (for which it has complete information). As discussed further in Section 9.7, many existing escape analyses either have or can easily be extended to have the other two key properties [172, 58, 35]. Many of these algorithms are significantly more efficient than our base algorithm, and we would expect incrementalization to provide these algorithms with additional efficiency increases comparable to those we observed for our algorithm. Compiler developers would therefore be able to choose from a variety of efficient analyses, with some analyses imposing little to no overhead.

An arguably more important benefit is the fact that incrementalized algorithms usually analyze only a local neighborhood of the program surrounding each object allocation site. The analysis time for each site is therefore independent of the overall size of the program, enabling the analysis to scale to handle programs of arbitrary size. And incrementalized algorithms can analyze incomplete programs.

9.1.5 Contributions

This paper makes the following contributions:

- **Analysis Approach:** It presents an incremental approach to program analysis. Instead of analyzing the entire program, the analysis is focused by an analysis policy to incrementally analyze only those regions of the program that may provide useful results.

- **Analysis Algorithm:** It presents a new combined pointer and escape analysis algorithm based on the incremental approach described above.

- **Analysis Policy:** It formulates the analysis policy as a solution to an investment problem. Presented with several analysis opportunities, the analysis policy incrementally invests analysis resources in those opportunities that offer the best estimated marginal return.

- **Experimental Results:** Our experimental results show that, for our benchmark programs, our analysis policy delivers almost all of the benefit of the whole-program analysis at a fraction of the cost.

The remainder of the paper is structured as follows. Section 9.2 presents several examples. Section 9.3 presents our previously published base whole-program analysis [202]; readers familiar with this analysis can skip this section. Section 9.4 presents the incrementalized analysis. Section 9.5 presents the analysis policy; Section 9.6 presents experimental results. Section 9.7 discusses related work; we conclude in Section 9.8.

9.2 Examples

We next present several examples that illustrate the basic approach of our analysis. Figure 9-1 presents two classes: the `complex` class, which implements a complex number package, and the `client` class, which uses the package. The `complex` class uses two mechanisms for returning values to callers: the `add` and `multiplyAdd` methods write the result into the receiver object (the `this` object), while the `multiply` method allocates a new object to hold the result.

9.2.1 The `compute` Method

We assume that the analysis policy first targets the object allocation site at line `3` of the `compute` method. The goal is to capture the objects allocated at this site and allocate them on the call stack. The initial analysis of `compute` skips the call to the `multiplyAdd` method. Because the analysis is flow sensitive, it produces a points-to escape graph for each program point in the `compute` method. Because the stack allocation optimization ties object lifetimes to method lifetimes, the legality of this optimization is determined by the points-to escape graph at the end of the method.

Figure 9-2 presents the points-to escape graph from the end of the `compute` method. The solid nodes are *inside* nodes, which represent objects created inside the currently analyzed region of the program. Node `3` is an inside node that represents all objects created at line `3` in the `compute` method. The dashed nodes are *outside* nodes, which represent objects not identified as created inside the analyzed region. Nodes `1` and `2` are a kind of outside node called a *parameter* node; they represent the parameters to the `compute` method. The analysis result also records the skipped call sites and the actual parameters at each site.

In this case, the analysis policy notices that the target node (node `3`) escapes because it is a parameter to the skipped call to `multiplyAdd`. It therefore directs

```
class complex {
  double x,y;
  complex(double a, double b) { x = a; y = b; }
  void add(complex u, complex v) {
    x = u.x+v.x; y = u.y+v.y;
  }
  complex multiply(complex m) {
11: complex r = new complex(x*m.x-y*m.y, x*m.y+y*m.x);
    return(r);
  }
  void multiplyAdd(complex a, complex b, complex c) {
    complex s = b.multiply(c);
    this.add(a, s);
  }
}
class client {
  public static void compute(complex d, complex e) {
3:  complex t = new complex(0.0, 0.0);
    t.multiplyAdd(d,e,e);
  }
}
```

Figure 9-1: Complex Number and Client Classes

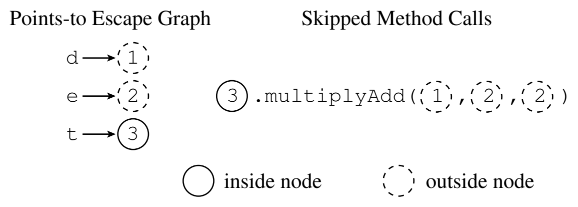


Figure 9-2: Analysis Result from compute Method

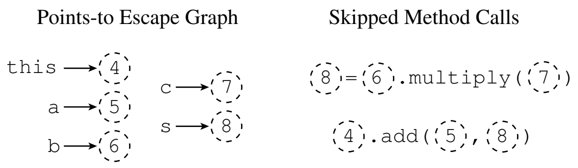


Figure 9-3: Analysis Result from multiplyAdd Method

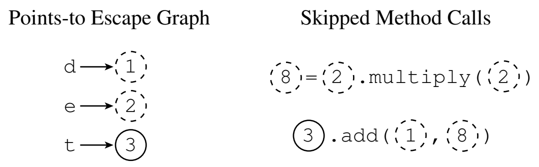


Figure 9-4: Analysis Result from compute Method after Integrating Result from multiplyAdd

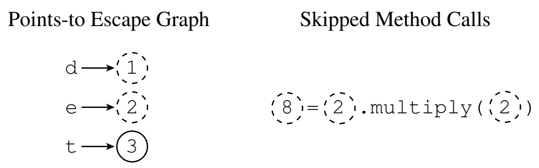


Figure 9-5: Analysis Result from `compute` Method after Integrating Results from `multiplyAdd` and `add`

the algorithm to analyze the `multiplyAdd` method and integrate the result into the points-to escape graph from the program point at the end of the `compute` method.

Figure 9-3 presents the points-to escape graph from the initial analysis of the `multiplyAdd` method. Nodes `4` through `7` are parameter nodes. Node `8` is another kind of outside node: a *return* node that represents the return value of an unanalyzed method, in this case the `multiply` method. To integrate this graph into the caller graph from the `compute` method, the analysis first maps the parameter nodes from the `multiplyAdd` method to the nodes that represent the actual parameters at the call site. In our example, node `4` maps to node `3`, node `5` maps to node `1`, and nodes `6` and `7` both map to node `2`. The analysis uses this mapping to combine the graphs into the new graph in Figure 9-4. The analysis policy examines the new graph and determines that the target node now escapes via the call to the `add` method. It therefore directs the algorithm to analyze the `add` method and integrate the resulting points-to escape graph into the current graph for the `compute` method. Note that because the call to the `multiply` method has no effect on the escape status of the target node, the analysis policy directs the algorithm to leave this method unanalyzed.

Figure 9-5 presents the new graph after the integration of the graph from the `add` method. Because the `add` method does not change the points-to or escape information, the net effect is simply to remove the skipped call to the `add` method. Note that the target node (node `3`) is captured in this graph, which implies that it is not accessible when the `compute` method returns. The compiler can therefore generate code that allocates all objects from the corresponding allocation site in the activation record of this method.

9.2.2 The `multiply` Method

The analysis next targets the object allocation site at line `11` of the `multiply` method in Figure 9-1. Figure 9-6 presents the points-to escape graph from this method, which indicates that the target node (node `11`) escapes to the caller (in this case the `multiplyAdd` method) via the return value. The algorithm avoids repeated method reanalyses by retrieving the cached points-to escape graph for the `multiplyAdd` method, then integrating the graph from the `multiply` method into this cached graph. Figure 9-7 presents the resulting points-to escape graph, which is cached as the new (more precise) points-to escape graph for the `multiplyAdd` method. This graph indicates that the target node (node `11`) does not escape to the caller of the `multiplyAdd`

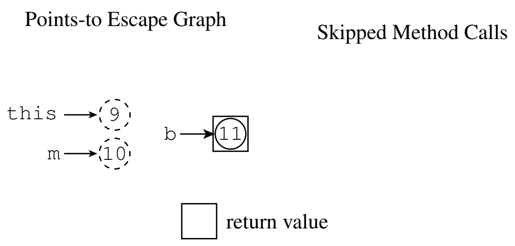


Figure 9-6: Analysis Result from `multiply` Method

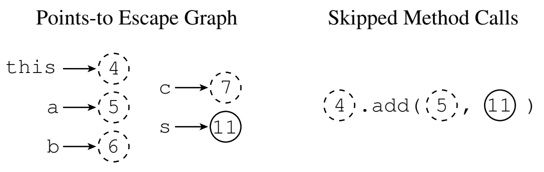


Figure 9-7: Analysis Result from `multiplyAdd` Method after Integrating Result from `multiply` Method

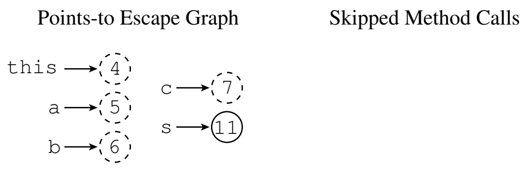


Figure 9-8: Analysis Result from `multiplyAdd` Method after Integrating Results from `multiply` and `add`

method, but does escape via the unanalyzed call to the `add` method. The analysis therefore retrieves the cached points-to escape graph from the `add` method, then integrates this graph into the current graph from the `multiplyAdd` method. Figure 9-8 presents the resulting graph. Once again, the algorithm caches this result as the new graph for the `multiplyAdd` method. The target node (node `11`) is captured in this graph — it escapes its enclosing method (the `multiply` method), but is recaptured in a caller (the `multiplyAdd` method).

At this point the compiler has several options: it can inline the `multiply` method into the `multiplyAdd` method and allocate the object on the stack, or it can preallocate the object on the stack frame of the `multiplyAdd` method, then pass it in by reference to a specialized version of the `multiply` routine. Both options enable stack allocation even if the node is captured in some but not all invocation paths, if the analysis policy declines to analyze all potential callers, or if it is not possible to identify all potential callers at compile time. Our implemented compiler uses inlining.

9.2.3 Object Field Accesses

Our next example illustrates how the analysis deals with object field accesses. Figure 9-9 presents a rational number class that deals with return values in yet another way. Each `Rational` object has a field called `result`; the methods in Figure 9-9 that operate on these objects store the result of their computation in this field for the caller to access.

```
class Rational {
  int numerator, denominator;
  Rational result;
  Rational(int n, int d) {
    numerator = n;
    denominator = d;
  }
  void scale(int m) {
    result = new Rational(numerator * m,
                denominator);
  }
  void abs() {
    int n = numerator;
    int d = denominator;
    if (n < 0) n = -n;
    if (d < 0) d = -d;
    if (d % n == 0) {
 4:   result = new Rational(n / d, 1);
    } else {
 5:   result = new Rational(n, d);
    }
  }
}
class client {
  public static void evaluate(int i, int j) {
1:  Rational r = new Rational(0.0, 0.0);
    r.abs();
2:  Rational n = r.result;
    n.scale(m);
  }
}
```

Figure 9-9: Rational Number and Client Classes

We next discuss how the analysis policy guides the analysis for the `Rational` allocation site at line `1` in the `evaluate` method. Figure 9-10 presents the initial analysis result at the end of this method. The dashed edge between nodes 1 and 2 is an *outside* edge, which represents references not identified as created inside the currently analyzed region of the program. Outside edges always point from an escaped node to a new kind of outside node, a *load* node, which represents objects whose references are loaded at a given load statement, in this case the statement `n = r.result` at line 2 in the `evaluate` method.

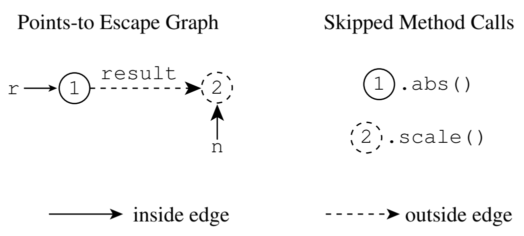


Figure 9-10: Analysis Result from `evaluate` Method

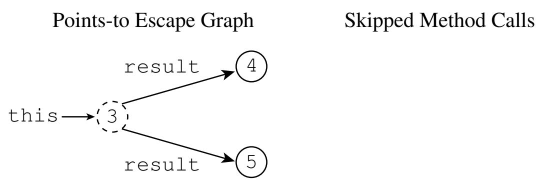


Figure 9-11: Analysis Result from `abs` Method

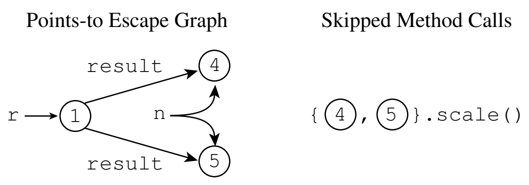


Figure 9-12: Analysis Result from `evaluate` After Integrating Result from `abs`

The analysis policy notices that the target node (node `1`) escapes via a call to the `abs` method. It therefore directs the analysis to analyze `abs` and integrate the result into the result from the end of the `evaluate` method. Figure 9-11 presents the analysis result from the end of the `abs` method. Node `3` represents the receiver object, node `4` represents the object created at line `4` of the `abs` method, and node `5` represents the object created at line `5`. The solid edges from node `3` to nodes `4` and `5` are *inside* edges. Inside edges represent references created within the analyzed region of the program, in this case the `abs` method.

The algorithm next integrates this graph into the analysis result from `evaluate`. The goal is to reconstruct the result of the base whole-program analysis. In the base analysis, which does not skip call sites, the analysis of `abs` changes the points-to escape graph at the program point after the call site. These changes in turn affect the analysis of the statements in `evaluate` after the call to `abs`. The incrementalized analysis reconstructs the analysis result as follows. It first determines that node `3` represented node `1` during the analysis of `abs`. It then matches the outside edge against the two inside edges to determine that, during the analysis of the region of `evaluate` after the skipped call to `abs`, the outside edge from node `1` to node `2` represented the inside edges from node `3` to nodes `4` and `5`, and that the load node `2` therefore represented nodes `4` and `5`. The combined graph therefore contains inside edges from node `1` to nodes `4` and `5`. Because node `1` is captured, the analysis removes the outside edge from this node. Finally, the updated analysis replaces the load node `2` in the skipped call site to `scale` with nodes `4` and `5`. At this point the analysis has captured node `1` inside the `evaluate` method, enabling the compiler to stack allocate all of the objects created at the corresponding allocation site at line `1` in Figure 9-9.

9.3 The Base Analysis

The base analysis is a previously published points-to and escape analysis [202]. For completeness, we present the algorithm again here. The algorithm is compositional, analyzing each method once before its callers to extract a single parameterized analysis result that can be specialized for use at different call sites.[1] It therefore analyzes the program in a bottom-up fashion from the leaves of the call graph towards the root. To simplify the presentation we ignore static class variables, exceptions, and return values. Our implemented algorithm correctly handles all of these features.

9.3.1 Object Representation

The analysis represents the objects that the program manipulates using a set $n \in N$ of nodes, which consists of a set N_I of inside nodes and a set N_O of outside nodes. Inside nodes represent objects created inside the currently analyzed region of the program, i.e., inside the current method or one of the analyzed methods that it (transitively)

[1]Recursive programs require a fixed-point algorithm that may analyze methods involved in cycles in the call graph multiple times.

invokes. There is one inside node for each object allocation site; that node represents all objects created at that site. The inside nodes include the set of thread nodes $N_T \subseteq N_I$. Thread nodes represent thread objects, i.e. objects that inherit from `Thread` or implement the `Runnable` interface.

The set of parameter nodes $N_P \subseteq N_O$ represents objects passed as parameters into the currently analyzed method. There is one load node $n \in N_L \subseteq N_O$ for each load statement in the program; that node represents all objects whose references are 1) loaded at that statement, and 2) not identified as created inside the currently analyzed region of the program. There is also a set $\mathtt{f} \in \mathtt{F}$ of fields in objects, a set $\mathtt{v} \in \mathtt{V}$ of local or parameter variables, and a set $\mathtt{l} \in \mathtt{L} \subseteq \mathtt{V}$ of local variables.

9.3.2 Points-To Escape Graphs

A points-to escape graph is a pair $\langle O, I \rangle$, where

- $O \subseteq (N \times \mathtt{F}) \times N_L$ is a set of outside edges. We write an edge $\langle \langle n_1, \mathtt{f} \rangle, n_2 \rangle$ as $n_1 \xrightarrow{\mathtt{f}} n_2$.

- $I \subseteq ((N \times \mathtt{F}) \times N) \cup (\mathtt{V} \times N)$ is a set of inside edges. We write an edge $\langle \mathtt{v}, n \rangle$ as $\mathtt{v} \rightarrow n$ and an edge $\langle \langle n_1, \mathtt{f} \rangle, n_2 \rangle$ as $n_1 \xrightarrow{\mathtt{f}} n_2$.

Inside edges represent references created within the currently analyzed part of the program. Outside edges represent references not identified as created within the currently analyzed part of the program. Outside edges usually represent references created outside the currently analyzed part of the program, but when multiple nodes represent the same object (for example, when a method is invoked with aliased parameters), an outside edge from one node can represent a reference from the object created within the currently analyzed part of the program.

A node escapes if it is reachable in $O \cup I$ from a parameter node or a thread node. We formalize this notion by defining an escape function

$$e_{O,I}(n) = \{n' \in N_T \cup N_P . n \text{ is reachable from } n' \text{ in } O \cup I\}$$

that returns the set of parameter and thread nodes through which n escapes. We define the concepts of escaped and captured nodes as follows:

- escaped$(\langle O, I \rangle, n)$ if $e_{O,I}(n) \neq \emptyset$

- captured$(\langle O, I \rangle, n)$ if $e_{O,I}(n) = \emptyset$

We say that an allocation site escapes or is captured in the context of a given analysis if the corresponding inside node is escaped or captured in the points-to escape graph that the analysis produces.

9.3.3 Program Representation

The algorithm represents the computation of each method using a control flow graph. We assume the program has been preprocessed so that all statements relevant to the analysis are either a copy statement $\mathtt{l} = \mathtt{v}$, a load statement $\mathtt{l}_1 = \mathtt{l}_2.\mathtt{f}$, a store statement $\mathtt{l}_1.\mathtt{f} = \mathtt{l}_2$, an object allocation statement $\mathtt{l} = \mathtt{new\ cl}$, or a method call statement $\mathtt{l}_0.\mathtt{op}(\mathtt{l}_1, \ldots, \mathtt{l}_k)$.

9.3.4 Intraprocedural Analysis

The intraprocedural analysis is a forward dataflow analysis that produces a points-to escape graph for each program point in the method. Each method is analyzed under the assumption that the parameters are *maximally unaliased*, i.e., point to different objects. For a method with formal parameters $\mathtt{v}_0, \ldots, \mathtt{v}_n$, the initial points-to escape graph at the entry point of the method is $\langle \emptyset, \{\langle \mathtt{v}_i, n_{\mathtt{v}_i} \rangle . 1 \leq i \leq n\} \rangle$ where $n_{\mathtt{v}_i}$ is the parameter node for parameter $\mathtt{v}_i$. If the method is invoked in a context where some of the parameters may point to the same object, the interprocedural analysis described below in Section 9.3.5 merges parameter nodes to conservatively model the effect of the aliasing.

The transfer function $\langle O', I' \rangle = [\![\mathtt{st}]\!](\langle O, I \rangle)$ models the effect of each statement $\mathtt{st}$ on the current points-to escape graph. Figure 9-13 graphically presents the rules that determine the new graph for each statement. Each row in this figure contains three items: a statement, a graphical representation of existing edges, and a graphical representation of the existing edges plus the new edges that the statement generates. Two of the rows (for statements $\mathtt{l}_1 = \mathtt{l}_2.\mathtt{f}$ and $\mathtt{l} = \mathtt{new\ cl}$) also have a where clause that specifies a set of side conditions. The interpretation of each row is that whenever the points-to escape graph contains the existing edges and the side conditions are satisfied, the transfer function for the statement generates the new edges. Assignments to a variable kill existing edges from that variable; assignments to fields of objects leave existing edges in place. At control-flow merges, the analysis takes the union of the inside and outside edges. At the end of the method, the analysis removes all captured nodes and local or parameter variables from the points-to escape graph.

9.3.5 Interprocedural Analysis

At each call statement, the interprocedural analysis uses the analysis result from each potentially invoked method to compute a transfer function for the statement. We assume a call site of the form $\mathtt{l}_0.\mathtt{op}(\mathtt{l}_1, \ldots, \mathtt{l}_k)$, a potentially invoked method $\mathtt{op}$ with formal parameters $\mathtt{v}_0, \ldots, \mathtt{v}_k$, a points-to escape graph $\langle O_1, I_1 \rangle$ at the program point before the call site, and a graph $\langle O_2, I_2 \rangle$ from the end of $\mathtt{op}$.

A map $\mu \subseteq N \times N$ combines the callee graph into the caller graph. The map serves two purposes: 1) it maps each outside node in the callee to the nodes in the caller that it represents during the analysis of the callee, and 2) it maps each node in the callee to itself if that node should be present in the combined graph. We use the notation $\mu(n) = \{n' . \langle n, n' \rangle \in \mu\}$ and $n_1 \xrightarrow{\mu} n_2$ for $n_2 \in \mu(n_1)$.

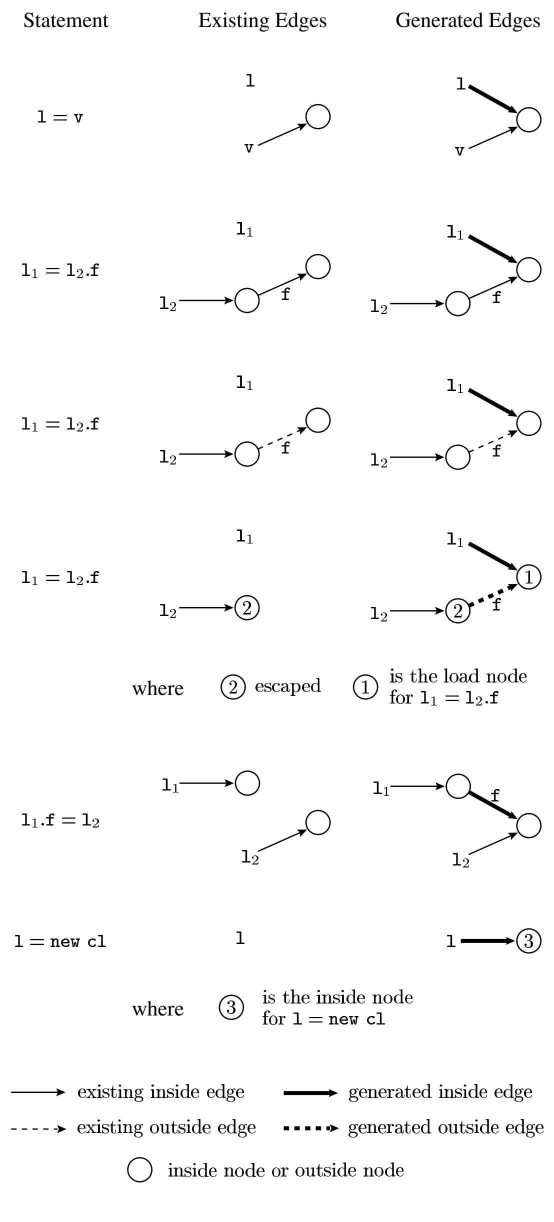


Figure 9-13: Generated Edges for Basic Statements

The interprocedural mapping algorithm $\langle\langle O, I\rangle, \mu\rangle =$ $\text{map}(\langle O_1, I_1\rangle, \langle O_2, I_2\rangle, \hat{\mu})$ starts with the points-to escape graph $\langle O_1, I_1\rangle$ from the caller, the graph $\langle O_2, I_2\rangle$ from the callee, and an initial parameter map

$$\hat{\mu}(n) = \begin{cases} I_1(\texttt{l}_i) & \text{if } \{n\} = I_2(\texttt{v}_i) \\ \emptyset & \text{otherwise} \end{cases}$$

that maps each parameter node from the callee to the nodes that represent the corresponding actual parameters at the call site. It produces the new mapped edges from the callee $\langle O, I\rangle$ and the new map μ.

Figure 9-14 presents the constraints that define the new edges $\langle O, I\rangle$ and new map μ. Constraint 9.1 initializes the map μ to the initial parameter map $\hat{\mu}$. Constraint 9.2 extends μ, matching outside edges from the callee against edges from the caller to ensure that μ maps each outside node from the callee to the corresponding nodes in the caller that it represents during the analysis of the callee. Constraint 9.3 extends μ to model situations in which aliasing in the caller causes an outside node from the callee to represent other callee nodes during the analysis of the callee. Constraints 9.4 and 9.5 complete the map by computing which nodes from the callee should be present in the caller and mapping these nodes to themselves. Constraints 9.6 and 9.7 use the map to translate inside and outside edges from the callee into the caller. The new graph at the program point after the call site is $\langle I_1 \cup I, O_1 \cup O\rangle$.

$$\hat{\mu}(n) \subseteq \mu(n) \tag{9.1}$$

$$\frac{n_1 \xrightarrow{\texttt{f}} n_2 \in O_2, n_3 \xrightarrow{\texttt{f}} n_4 \in O_1 \cup I_1, n_1 \xrightarrow{\mu} n_3}{n_2 \xrightarrow{\mu} n_4} \tag{9.2}$$

$$\frac{\begin{array}{c} n_1 \xrightarrow{\mu} n_3, n_2 \xrightarrow{\mu} n_3, n_1 \neq n_2, \\ n_1 \xrightarrow{\texttt{f}} n_4 \in O_2, n_2 \xrightarrow{\texttt{f}} n_5 \in O_2 \cup I_2 \end{array}}{\mu(n_4) \subseteq \mu(n_5)} \tag{9.3}$$

$$\frac{n_1 \xrightarrow{\texttt{f}} n_2 \in I_2, n_1 \xrightarrow{\mu} n, n_2 \in N_I}{n_2 \xrightarrow{\mu} n_2} \tag{9.4}$$

$$\frac{n_1 \xrightarrow{\texttt{f}} n_2 \in O_2, n_1 \xrightarrow{\mu} n, \text{escaped}(\langle O, I\rangle, n)}{n_2 \xrightarrow{\mu} n_2} \tag{9.5}$$

$$\frac{n_1 \xrightarrow{\texttt{f}} n_2 \in I_2}{(\mu(n_1) \times \{\texttt{f}\}) \times \mu(n_2) \subseteq I} \tag{9.6}$$

$$\frac{n_1 \xrightarrow{\texttt{f}} n_2 \in O_2, n_2 \xrightarrow{\mu} n_2}{(\mu(n_1) \times \{\texttt{f}\}) \times \{n_2\} \subseteq O} \tag{9.7}$$

Figure 9-14: Constraints for Interprocedural Analysis

Because of dynamic dispatch, a single call site may invoke several different methods. The transfer function therefore merges the points-to escape graphs from the analysis of all potentially invoked methods to derive the new graph at the point after the call site. The current implementation obtains this call graph information using a variant of a cartesian product type analysis [1], but it can use any conservative approximation to the dynamic call graph.

9.3.6 Merge Optimization

As presented so far, the analysis may generate points-to escape graphs $\langle O, I\rangle$ in which a node n may have multiple distinct outside edges $n \xrightarrow{\mathtt{f}} n_1, \ldots, n \xrightarrow{\mathtt{f}} n_k \in O$. We eliminate this inefficiency by merging the load nodes $n_1, \ldots, n_k$. With this optimization, a single load node may be associated with multiple load statements. The load node generated from the merge of k load nodes $n_1, \ldots, n_k$ is associated with all of the statements of $n_1, \ldots, n_k$.

9.4 The Incrementalized Analysis

We next describe how to *incrementalize* the base algorithm — how to enhance the algorithm so that it can skip the analysis of call sites while maintaining enough information to reconstruct the result of analyzing the invoked methods should the analysis policy direct the analysis to do so. The first step is to record the set S of skipped call sites. For each skipped call site s, the analysis records the invoked method $\mathtt{op}_s$ and the initial parameter map $\hat{\mu}_s$ that the base algorithm would compute at that call site. To simplify the presentation, we assume that each skipped call site is 1) executed at most once, and 2) invokes a single method. Section 9.4.8 discusses how we eliminate these restrictions in our implemented algorithm.

The next step is to define an updated escape function $e_{S,O,I}$ that determines how objects escape the currently analyzed region of the program via skipped call sites:

$$\begin{aligned} e_{S,O,I}(n) = \{s \in S.\exists n_1 \in N_P.n_1 \xrightarrow{\hat{\mu}_s} n_2 \text{ and} \\ n \text{ is reachable from } n_2 \text{ in } O \cup I\} \cup e_{O,I}(n) \end{aligned}$$

We adapt the interprocedural mapping algorithm from Section 9.3.5 to use this updated escape function. By definition, n escapes through a call site s if $s \in e_{S,O,I}(n)$.

A key complication is preserving flow sensitivity with respect to previously skipped call sites during the integration of analysis results from those sites. For optimization purposes, the compiler works with the analysis result from the end of the method. But the skipped call sites occur at various program points inside the method. We therefore augment the points-to escape graphs from the base analysis with several orders, which record ordering information between edges in the points-to escape graph and skipped call sites:

- $\omega \subseteq S \times ((N \times \{\mathtt{f}\}) \times N_L)$. For each call site s, $\omega(s) = \{n_1 \xrightarrow{\mathtt{f}} n_2. \left\langle s, n_1 \xrightarrow{\mathtt{f}} n_2 \right\rangle \in \omega\}$ is the set of outside edges that the analysis generates before it skips s.

- $\iota \subseteq S \times ((N \times \{\mathtt{f}\}) \times N)$. For each call site s, $\iota(s) = \{n_1 \xrightarrow{\mathtt{f}} n_2 .\left\langle s, n_1 \xrightarrow{\mathtt{f}} n_2 \right\rangle \in \iota\}$ is the set of inside edges that the analysis generates before it skips s.

- $\tau \subseteq S \times ((N \times \{\mathtt{f}\}) \times N_L)$. For each call site s, $\tau(s) = \{n_1 \xrightarrow{\mathtt{f}} n_2 .\left\langle s, n_1 \xrightarrow{\mathtt{f}} n_2 \right\rangle \in \tau\}$ is the set of outside edges that the analysis generates after it skips s.

- $\nu \subseteq S \times ((N \times \{\mathtt{f}\}) \times N)$. For each call site s, $\nu(s) = \{n_1 \xrightarrow{\mathtt{f}} n_2 .\left\langle s, n_1 \xrightarrow{\mathtt{f}} n_2 \right\rangle \in \nu\}$ is the set of inside edges that the analysis generates after it skips s.

- $\beta \subseteq S \times S$. For each call site s, $\beta(s) = \{s' . \langle s, s' \rangle \in \beta\}$ is the set of call sites that the analysis skips before skipping s.

- $\alpha \subseteq S \times S$. For each call site s, $\alpha(s) = \{s' . \langle s, s' \rangle \in \alpha\}$ is the set of call sites that the analysis skips after skipping s.

The incrementalized analysis works with augmented points-to escape graphs of the form $\langle O, I, S, \omega, \iota, \tau, \nu, \beta, \alpha \rangle$. Note that because β and α are inverses,[2] the analysis does not need to represent both explicitly. It is of course possible to use any conservative approximation of ω, ι, τ, ν, β and α; an especially simple approach uses $\omega(s) = \tau(s) = O$, $\iota(s) = \nu(s) = I$, and $\beta(s) = \alpha(s) = S$.

We next discuss how the analysis uses these additional components during the incremental analysis of a call site. We assume a current augmented points-to escape graph
$\langle O_1, I_1, S_1, \omega_1, \iota_1, \tau_1, \nu_1, \beta_1, \alpha_1 \rangle$, a call site $s \in S_1$ with invoked operation $\mathtt{op}_s$, and an augmented points-to escape graph $\langle O_2, I_2, S_2, \omega_2, \iota_2, \tau_2, \nu_2, \beta_2, \alpha_2 \rangle$ from the end of $\mathtt{op}_s$.

9.4.1 Matched Edges

In the base algorithm, the analysis of a call site matches outside edges from the analyzed method against existing edges in the points-to escape graph from the program point before the site. By the time the algorithm has propagated the graph to the end of the method, it may contain additional edges generated by the analysis of statements that execute after the call site. When the incrementalized algorithm integrates the analysis result from a skipped call site, it matches outside edges from the invoked method against only those edges that were present in the points-to escape graph at the program point before the call site. $\omega(s)$ and $\iota(s)$ provide just those edges. The algorithm therefore computes

$$\langle O, I, \mu \rangle = \mathrm{map}(\langle \omega_1(s), \iota_1(s) \rangle, \langle O_2, I_2 \rangle, \hat{\mu}_s)$$

where O and I are the new sets of edges that the analysis of the callee adds to the caller graph.

[2] Under the interpretation $\beta^{-1} = \{\langle s_1, s_2 \rangle . \langle s_2, s1 \rangle \in \beta\}$ and $\alpha^{-1} = \{\langle s_1, s_2 \rangle . \langle s_2, s1 \rangle \in \alpha\}$, $\beta = \alpha^{-1}$ and $\beta^{-1} = \alpha$.

9.4.2 Propagated Edges

In the base algorithm, the transfer function for an analyzed call site may add new edges to the points-to graph from before the site. These new edges create effects that propagate through the analysis of subsequent statements. Specifically, the analysis of these subsequent statements may read the new edges, then generate additional edges involving the newly referenced nodes. In the points-to graph from the incrementalized algorithm, the edges from the invoked method will not be present if the analysis skips the call site. But these missing edges must come (directly or indirectly) from nodes that escape into the skipped call site. In the points-to graphs from the caller, these missing edges are represented by outside edges that are generated by the analysis of subsequent statements. The analysis can therefore use $\tau_1(s)$ and $\nu_1(s)$ to reconstruct the propagated effect of analyzing the skipped method. It computes

$$\langle O', I', \mu' \rangle = \text{map}(\langle O, I \rangle, \langle \tau_1(s), \nu_1(s) \rangle, \{\langle n, n \rangle .n \in N\})$$

where O' and I' are the new sets of edges that come from the interaction of the analysis of the skipped method and subsequent statements, and μ' maps each outside node from the caller to the nodes from the callee that it represents during the analysis from the program point after the skipped call site to the end of the method. Note that this algorithm generates all of the new edges that a complete reanalysis would generate. But it generates the edges incrementally without reanalyzing the code.

9.4.3 Skipped Call Sites from the Caller

In the base algorithm, the analysis of one call site may affect the initial parameter map for subsequent call sites. Specifically, the analysis of a site may cause the formal parameter nodes at subsequent sites to be mapped to additional nodes in the graph from the caller.

For each skipped call site, the incrementalized algorithm records the parameter map that the base algorithm would have used at that site. When the incrementalized algorithm integrates an analysis result from a previously skipped site, it must update the recorded parameter maps for subsequent skipped sites. At each of these sites, outside nodes represent the additional nodes that the analysis of the previously skipped site may add to the map. And the map μ' records how each of these outside nodes should be mapped. For each subsequent site $s' \in \alpha_1(s)$, the algorithm composes the site's current recorded parameter map $\hat{\mu}_{s'}$ with μ' to obtain its new recorded parameter map $\mu' \circ \hat{\mu}_{s'}$.

9.4.4 Skipped Call Sites from the Callee

The new set of skipped call sites $S' = (S_1 \cup S_2)$ contains the set of skipped call sites S_2 from the callee. When it maps the callee graph into the caller graph, the analysis updates the recorded parameter maps for the skipped call sites in S_2. For each site $s' \in S_2$, the analysis simply composes the site's current map $\hat{\mu}_{s'}$ with the map μ to obtain the new recorded parameter map $\mu \circ \hat{\mu}_{s'}$ for s'.

9.4.5 New Orders

The analysis constructs the new orders by integrating the orders from the caller and callee into the new analysis result and extending the orders for s to the mapped edges and skipped call sites from the callee. So, for example, the new order between outside edges and subsequent call sites (ω') consists of the order from the caller (ω_1), the mapped order from the callee ($\omega_2[\mu]$), the order from s extended to the skipped call sites from the callee ($S_2 \times \omega_1(s)$), and the outside edges from the callee ordered with respect to the call sites after s ($\alpha_1(s) \times O$):

$$\begin{aligned}
\omega' &= \omega_1 \cup \omega_2[\mu] \cup (S_2 \times \omega_1(s)) \cup (\alpha_1(s) \times O) \\
\iota' &= \iota_1 \cup \iota_2[\mu] \cup (S_2 \times \iota_1(s)) \cup (\alpha_1(s) \times I) \\
\tau' &= \tau_1 \cup \tau_2[\mu] \cup (S_2 \times \tau_1(s)) \cup (\beta_1(s) \times O) \\
\nu' &= \nu_1 \cup \nu_2[\mu] \cup (S_2 \times \nu_1(s)) \cup (\beta_1(s) \times I) \\
\beta' &= \beta_1 \cup \beta_2 \cup (S_2 \times \beta_1(s)) \cup (\alpha_1(s) \times S_2) \\
\alpha' &= \alpha_1 \cup \alpha_2 \cup (S_2 \times \alpha_1(s)) \cup (\beta_1(s) \times S_2)
\end{aligned}$$

Here $\omega[\mu]$ is the order ω under the map μ, i.e., $\omega[\mu] = \{\left\langle s, n_1' \xrightarrow{\mathtt{f}} n_2' \right\rangle . \left\langle s, n_1 \xrightarrow{\mathtt{f}} n_2 \right\rangle \in \omega, n_1 \xrightarrow{\mu} n_1',$ and $n_2 \xrightarrow{\mu} n_2'\}$, and similarly for ι, τ, and ν.

9.4.6 Cleanup

At this point the algorithm can compute a new graph $\langle O_1 \cup O \cup O', I_1 \cup I \cup I', S', \omega', \iota', \tau', \nu', \beta', \alpha' \rangle$ that reflects the integration of the analysis of s into the previous analysis result $\langle O_1, I_1, S_1, \omega_1, \iota_1, \tau_1, \nu_1, \beta_1, \alpha_1 \rangle$. The final step is to remove s from all components of the new graph and to remove all outside edges from captured nodes.

9.4.7 Updated Intraprocedural Analysis

The transfer function for a skipped call site s performs the following additional tasks:

- Record the initial parameter map $\hat{\mu}_s$ that the base algorithm would use when it analyzed the site.
- Update ω to include $\{s\} \times O$, update ι to include $\{s\} \times I$, update α to contain $S \times \{s\}$, and update β to contain $\{s\} \times S$.
- Update S to include the skipped call site s.

Whenever a load statement generates a new outside edge $n_1 \xrightarrow{\mathtt{f}} n_2$, the transfer function updates τ to include $S \times \{n_1 \xrightarrow{\mathtt{f}} n_2\}$. Whenever a store statement generates a new inside edge $n_1 \xrightarrow{\mathtt{f}} n_2$, the transfer function updates ν to include $S \times \{n_1 \xrightarrow{\mathtt{f}} n_2\}$.

Finally, the incrementalized algorithm extends the confluence operator to merge the additional components. For each additional component (including the recorded parameter maps μ_s), the confluence operator is set union.

9.4.8 Extensions

So far, we have assumed that each skipped call site is executed at most once and invokes a single method. We next discuss how our implemented algorithm eliminates these restrictions. To handle dynamic dispatch, we compute the graph for all of the possible methods that the call site may invoke, then merge these graphs to obtain the new graph.

We also extend the abstraction to handle skipped call sites that are in loops or are invoked via multiple paths in the control flow graph. We maintain a multiplicity flag for each call site specifying whether the call site may be executed multiple times:

- The transfer function for a skipped call site s checks to see if the site is already in the set of skipped sites S. If so, it sets the multiplicity flag to indicate that s may be invoked multiple times. It also takes the union of the site's current recorded parameter map $\hat{\mu}_s$ and the parameter map $\hat{\mu}$ from the transfer function to obtain the site's new recorded parameter map $\hat{\mu}_s \cup \hat{\mu}$.

- The algorithm that integrates analysis results from previously skipped call sites performs a similar set of operations to maintain the recorded parameter maps and multiplicity flags for call sites that may be present in the analysis results from both the callee and the caller. If the skipped call site may be executed multiple times, the analysis uses a fixed-point algorithm when it integrates the analysis result from the skipped call site. This algorithm models the effect of executing the site multiple times.

9.4.9 Recursion

The base analysis uses a fixed-point algorithm to ensure that it terminates in the presence of recursion. It is possible to use a similar approach in the incrementalized algorithm. Our implemented algorithm, however, does not check for recursion as it explores the call graph. If a node escapes into a recursive method, the analysis may, in principle, never terminate. In practice, the algorithm relies on the analysis policy to react to the expansion of the analyzed region by directing analysis resources to other allocation sites.

9.4.10 Incomplete Call Graphs

Our algorithm deals with incomplete call graphs as follows. If it is unable to locate all of the potential callers of a given method, it simply analyzes those it is able to locate. If it is unable to locate all potential callees at a given call site, it simply considers all nodes that escape into the site as permanently escaped.

9.5 Analysis Policy

The goal of the analysis policy is to find and analyze allocation sites that can be captured quickly and have a large optimization payoff. Conceptually, the policy uses the following basic approach. It estimates the payoff for capturing an allocation site as the number of objects allocated at that site in a previous profiling run. It uses empirical data and the current analysis result for the site to estimate the likelihood that it will ever be able to capture the site, and, assuming that it is able to capture the site, the amount of time required to do so. It then uses these estimates to calculate an estimated marginal return for each unit of analysis time invested in each site.

At each analysis step, the policy is faced with a set of partially analyzed sites that it can invest in. The policy simply chooses the site with the best estimated marginal return, and invests a (configurable) unit of analysis time in that site. During this time, the algorithm repeatedly selects one of the skipped call sites through which the allocation site escapes, analyzes the methods potentially invoked at that site (reusing the cached results if they are available), and integrates the results from these methods into the current result for the allocation site. If these analyses capture the site, the policy moves on to the site with the next best estimated marginal return. Otherwise, when the time expires, the policy recomputes the site's estimated marginal return in light of the additional information it has gained during the analysis, and once again invests in the (potentially different) site with the current best estimated marginal return.

9.5.1 Stack Allocation

The compiler applies two potential stack allocation optimizations depending on where an allocation site is captured:

- **Stack Allocate:** If the site is captured in the method that contains it, the compiler generates code to allocate all objects created at that site in the activation record of the containing method.

- **Inline and Stack Allocate:** If the site is captured in a direct caller of the method containing the site, the compiler first inlines the method into the caller. After inlining, the caller contains the site, and the generated code allocates all objects created at that site in the activation record of the caller.

The current analysis policy assumes that the compiler is 1) unable to inline a method if, because of dynamic dispatch, the corresponding call site may invoke multiple methods, and 2) unwilling to enable additional optimizations by further inlining the callers of the method containing the allocation site into their callers. It is, of course, possible to relax these assumptions to support more sophisticated inlining and/or specialization strategies.

Inlining complicates the conceptual analysis policy described above. Because each call site provides a distinct analysis context, the same allocation site may have different analysis characteristics and outcomes when its enclosing method is inlined at different call sites. The policy therefore treats each distinct combination of call site and allocation site as its own separate analysis opportunity.

9.5.2 Analysis Opportunities

The policy represents an opportunity to capture an allocation site a in its enclosing method `op` as $\langle a, \texttt{op}, G, p, c, d, m\rangle$, where G is the current augmented points-to escape graph for the site, p is the estimated payoff for capturing the site, c is the count of the number of skipped call sites in G through which a escapes, d is the method call depth of the analyzed region represented by G, and m is the mean cost of the call site analyses performed so far on behalf of this analysis opportunity. Note that a, `op`, and G are used to perform the incremental analysis, while p, c, d, and m are used to estimate the marginal return. Opportunities to capture an allocation site a in the caller `op` of its enclosing method have the form $\langle a, \texttt{op}, s, G, p, c, d, m\rangle$, where s is the call site in `op` that invokes the method containing a, and the remainder of the fields have the same meaning as before.

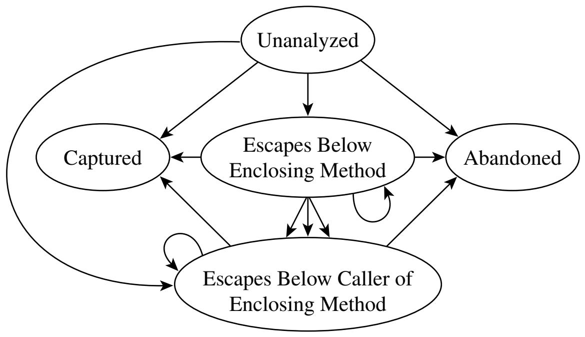

Unanalyzed
Captured
Escapes Below Enclosing Method
Abandoned
Escapes Below Caller of Enclosing Method

Figure 9-15: State-Transition Diagram for Analysis Opportunities

Figure 9-15 presents the state-transition diagram for analysis opportunities. Each analysis opportunity can be in one of the states of the diagram; the transitions correspond to state changes that take place during the analysis of the opportunity. The states have the following meanings:

- **Unanalyzed:** No analysis done on the opportunity.
- **Escapes Below Enclosing Method:** The opportunity's allocation site escapes into one or more skipped call sites, but does not (currently) escape to the caller of the enclosing method. The opportunity is of the form $\langle a, \texttt{op}, G, p, c, d, m\rangle$.
- **Escapes Below Caller of Enclosing Method:** The opportunity's site escapes to the caller of its enclosing method, but does not (currently) escape from this caller. The site may also escape into one or more skipped call sites. The opportunity is of the form $\langle a, \texttt{op}, s, G, p, c, d, m\rangle$.
- **Captured:** The opportunity's site is captured.
- **Abandoned:** The policy has permanently abandoned the analysis of the opportunity, either because its allocation site permanently escapes via a static

class variable or thread, because the site escapes to the caller of the caller of its enclosing method (and is therefore unoptimizable), or because the site escapes to the caller of its enclosing method and (because of dynamic dispatch) the compiler is unable to inline the enclosing method into the caller.

In Figure 9-15 there are multiple transitions from the Escapes Below Enclosing Method state to the Escapes Below Caller of Enclosing Method state. These transitions indicate that one Escapes Below Enclosing Method opportunity may generate multiple new Escapes Below Caller of Enclosing Method opportunities — one new opportunity for each potential call site that invokes the enclosing method from the old opportunity.

When an analysis opportunity enters the Escapes Below Caller of Enclosing Method state, the first analysis action is to integrate the augmented points-to escape graph from the enclosing method into the graph from the caller of the enclosing method.

9.5.3 Estimated Marginal Returns

If the opportunity is Unanalyzed, the estimated marginal return is $(\xi \cdot p)/\sigma$, where ξ is the probability of capturing an allocation site given no analysis information about the site, p is the payoff of capturing the site, and, assuming the analysis eventually captures the site, σ is the expected analysis time required to do so.

If the opportunity is in the state Escapes Below Enclosing Method, the estimated marginal return is $(\xi_1(d) \cdot p)/(c \cdot m)$. Here $\xi_1(d)$ is the conditional probability of capturing an allocation site given that the algorithm has explored a region of call depth d below the method containing the site, the algorithm has not (yet) captured the site, and the site has not escaped (so far) to the caller of its enclosing method. If the opportunity is in the state Escapes Below Caller of Enclosing Method, the estimated marginal return is $(\xi_2(d) \cdot p)/(c \cdot m)$. Here $\xi_2(d)$ has the same meaning as $\xi_1(d)$, except that the assumption is that the site has escaped to the caller of its enclosing method, but not (so far) to the caller of the caller of its enclosing method.

We obtain the capture probability functions ξ, ξ_1, and ξ_2 empirically by preanalyzing all of the executed allocation sites in some sample programs and collecting data that allows us to compute these functions. For Escapes Below Enclosing Method opportunities, the estimated payoff p is the number of objects allocated at the opportunity's allocation site a during a profiling run. For Escapes Below Caller of Enclosing Method opportunities, the estimated payoff is the number of objects allocated at the opportunity's allocation site a when the allocator is invoked from the opportunity's call site s.

When an analysis opportunity changes state or increases its method call depth, its estimated marginal return may change significantly. The policy therefore recomputes the opportunity's return whenever one of these events happens. If the best opportunity changes because of this recomputation, the policy redirects the analysis to work on the new best opportunity.

9.5.4 Termination

In principle, the policy can continue the analysis indefinitely as it invests in ever less profitable opportunities. In practice, it is important to terminate the analysis when the prospective returns become small compared to the analysis time required to realize them. We say that the analysis has *decided* an object if that object's opportunity is in the Captured or Abandoned state. The payoffs p in the analysis opportunities enable the policy to compute the current number of decided and undecided objects.

Two factors contribute to our termination policy: the percentage of undecided objects (this percentage indicates the maximum potential payoff from continuing the analysis), and the rate at which the analysis has recently been deciding objects. The results in Section 9.6 are from analyses terminated when the percentage of decided objects rises above 90% and the decision rate for the last quarter of the analysis drops below 1 percent per second, with a cutoff of 75 seconds of total analysis time.

We anticipate the development of a variety of termination policies to fit the particular needs of different compilers. A dynamic compiler, for example, could accumulate an analysis budget as a percentage of the time spent executing the application — the longer the application ran, the more time the policy would be authorized to invest analyzing it. The accumulation rate would determine the maximum amortized analysis overhead.

9.6 Experimental Results

We have implemented our analysis and the stack allocation optimization in the MIT Flex compiler, an ahead-of-time compiler written in Java for Java.[3] We ran the experiments on an 800 MHz Pentium III PC with 768Mbytes of memory running Linux Version 2.2.18. We ran the compiler using the Java Hotspot Client VM version 1.3.0 for Linux. The compiler generates portable C code, which we compile to an executable using gcc. The generated code manages the heap using the Boehm-Demers-Weiser conservative garbage collector [36] and uses `alloca` for stack allocation.

9.6.1 Benchmark Programs

Our benchmark programs include two multithreaded scientific computations (Barnes and Water), Jlex, and several Spec benchmarks (Db, Compress, and Raytrace). Barnes and Water are well-known benchmarks in the parallel computing community; our Java versions were derived from versions in the SPLASH-2 benchmark suite [205]. Figure 9-16 presents the compile and whole-program analysis times for the applications.

[3]The compiler is available at www.flexc.lcs.mit.edu.

Application	Compile Time Without Analysis	Whole-Program Analysis Time
Barnes	89.7	34.3
Water	91.1	38.2
Jlex	119.5	222.8
Db	93.6	126.6
Raytrace	118.4	102.2
Compress	219.6	645.1

Figure 9-16: Compile and Whole-Program Analysis Times (seconds)

9.6.2 Marginal Returns and Profiling Information

We derived the estimated capture probability functions ξ, ξ_1, and ξ_2 from an instrumented analysis of all of the executed object allocation sites in Barnes, Water, Db, and Raytrace. Figure 9-17 presents the capture probabilities $\xi_1(d)$ and $\xi_2(d)$ as a function of the call depth d; ξ is .33.

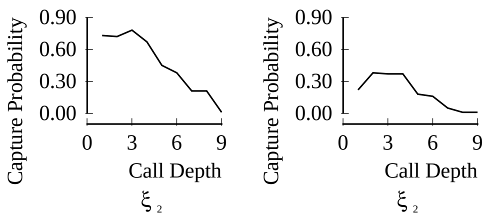


Figure 9-17: Capture Probability Functions

To compute the estimated marginal returns and implement the termination policy, the analysis policy needs an estimated optimization payoff for each allocation site. We obtain these payoffs as the number of objects allocated at each site during a training run on a small training input. The presented execution and analysis statistics are for executions on larger production inputs.

9.6.3 Analysis Payoffs and Statistics

Figure 9-18 presents analysis statistics from the incrementalized analysis. We present the number of captured allocation sites as the sum of two counts. The first count is the number of sites captured in the enclosing method; the second is the number captured in the caller of the enclosing method. Fractional counts indicate allocation sites that were captured in some but not all callers of the enclosing method. In Db, for example, one of the allocation sites is captured in two of the eight callers of its enclosing method. The Undecided Allocation Sites column counts the number of allocation sites in which the policy invested some resources, but did not determine whether it could stack allocate the corresponding objects or not. The Analyzed Call

	Analysis Time (seconds)	Captured Allocation Sites	Abandoned Allocation Sites	Undecided Allocation Sites	Total Allocation Sites	Analyzed Call Sites	Total Call Sites	Analyzed Methods	Total Methods
Barnes	0.8	3+0	0	2	736	18	1675	13	512
Water	21.7	33+0	4	33	748	94	1799	33	481
Jlex	0.9	0+2	1	2	1054	27	2879	12	569
Db	4.5	1+0.25	4	1.75	1118	54	2444	25	631
Raytrace	76.3	8+0.37	20.63	54	1067	271	3109	64	699
Compress	79.5	4+0.33	4	19.66	1354	111	4084	40	808

Figure 9-18: Analysis Statistics from Incrementalized Analysis

Sites, Total Call Sites, Analyzed Methods, and Total Methods columns show that the policy analyzes a small fraction of the total program.

The graphs in Figure 9-19 present three curves for each application. The horizontal dotted line indicates the percentage of objects that the whole-program analysis allocates on the stack. The dashed curve plots the percentage of decided objects (objects whose analysis opportunities are either Captured or Abandoned) as a function of the analysis time. The solid curve plots the percentage of objects allocated on the stack. For Barnes, Jlex, and Db, the incrementalized analysis captures virtually the same number of objects as the whole-program analysis, but spends a very small fraction of the whole-program analysis time to do so. Incrementalization provides less of a benefit for Water because two large methods account for a much of the analysis time of both analyses. For Raytrace and Compress, a bug in the 1.3 JVM forced us to run the incrementalized analysis, but not the whole-program analysis, on the 1.2 JVM. Our experience with the other applications indicates that both analyses run between five and six times faster on the 1.3 JVM than on the 1.2 JVM.

9.6.4 Application Execution Statistics

Figure 9-20 presents the total amount of memory that the applications allocate in the heap. Almost all of the allocated memory in Barnes and Water is devoted to temporary arrays that hold the results of intermediate computations. The C++ version of these applications allocates these arrays on the stack; our analysis restores this allocation strategy in the Java version. Most of the memory in Jlex is devoted to temporary iterators, which are stack allocated after inlining. Note the anomaly in Db and Compress: many objects are allocated on the stack, but the heap allocated objects are much bigger than the stack allocated objects.

Figure 9-21 presents the execution times. The optimizations provide a significant performance benefit for Barnes and Water and some benefit for Jlex and Db. Without stack allocation, Barnes and Water interact poorly with the conservative garbage collector. We expect that a precise garbage collector would reduce the performance difference between the versions with and without stack allocation.

9.7 Related Work

We first address related work in escape analysis, focusing on the prospects for incrementalizing existing algorithms. We then discuss several interprocedural analyses

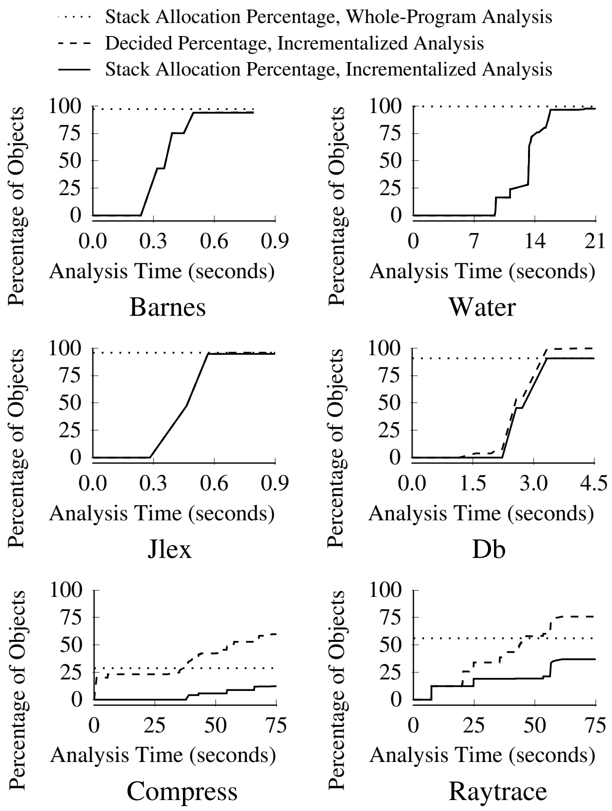


Figure 9-19: Analysis Time Payoffs

(demand-driven analysis, fragment analysis, and incremental analysis) that are designed to analyze part, but not all, of the program.

9.7.1 Escape Analysis

Many other researchers have developed escape analyses for Java [202, 58, 172, 35, 37]. These analyses have been presented as whole-program analyses, although many contain elements that make them amenable to incrementalization. All of the analyses listed above except the last [37] analyze methods independently of their callers, generating a summary that can be specialized for use at each call site. Unlike our base analysis [202], these analyses are not designed to skip call sites. But we believe it would be relatively straightforward to augment them to do so. With this extension in place, the remaining question is incrementalization. For flow-sensitive analyses [202, 58], the incrementalized algorithm must record information about the ordering of skipped call sites relative to the rest of the analysis information if it is to preserve the precision of the base whole-program analysis with respect to skipped call sites. Flow-insensitive analyses [172, 35], can ignore this ordering information and should therefore be able to use an extended abstraction that records only the mapping information for skipped call sites. In this sense flow-insensitive analyses should be, in general, simpler to incrementalize than flow-sensitive analyses.

Application	No Analysis	Incrementalized Analysis	Whole-Program Analysis
Barnes	36.0	3.2	2.0
Water	190.2	2.2	0.6
Jlex	40.8	3.1	2.5
Db	77.6	31.2	31.2
Raytrace	13.4	9.0	6.7
Compress	110.1	110.1	110.1

Figure 9-20: Allocated Heap Memory (Mbytes)

Application	No Analysis	Incrementalized Analysis	Whole-Program Analysis
Barnes	33.4	22.7	24.0
Water	18.8	11.2	10.7
Jlex	5.5	5.0	4.7
Db	103.8	104.0	101.3
Raytrace	3.0	2.9	2.9
Compress	44.9	44.8	45.1

Figure 9-21: Execution Times (seconds)

Escape analyses have typically been used for stack allocation and synchronization elimination. Our results show that analyzing a local region around each allocation site works well for stack allocation, presumably because stack allocation ties object lifetimes to the lifetimes of the capturing methods. But for synchronization elimination, a whole-program analysis may deliver significant additional optimization opportunities. For example, Ruf's synchronization elimination analysis determines which threads may synchronize on which objects [172]. In many cases, the analysis is able to determine that only one thread synchronizes on a given object, even though the object may be accessible to multiple threads or even, via a static class variable, to all threads. Exploiting this global information significantly improves the ability of the compiler to eliminate superfluous synchronization operations, especially for single threaded programs. We do not see how an incrementalized analysis could extract this kind of global information without scanning all of the code in each thread.

9.7.2 Demand-Driven Analysis

Demand-driven algorithms analyze only those parts of the program required to compute an analysis fact at a subset of the program points or to answer a given query [6, 147, 85, 164]. This approach can dramatically reduce the analyzed part of the program, providing a corresponding decrease in the analysis time. Like demand-driven analyses, our analysis does not analyze those parts of the program that do not affect the desired analysis results. Our approach differs in that it is designed to temporarily skip parts of the program even if the skipped parts potentially affect the analysis result. This approach works for its intended application (stack allocation) because it enables the analysis to choose from a set of potential optimization opportunities, some or all of which it is willing to forgo if the analysis cost is too high. In this context, avoiding excessively expensive, even if ultimately successful, analyses is as important as analyzing only those parts of the program required to obtain a specific result. Because our analysis can skip call sites, it can incrementally invest in multiple optimization opportunities, use the acquired information to improve its estimates of the marginal return of each opportunity, then dynamically redirect analysis resources to the currently most promising opportunities. In practice, this approach enables our analysis policy to quickly discover and exploit the best opportunities while avoiding opportunities that provide little or no optimization payoff.

9.7.3 Fragment and Incremental Analysis

Fragment analysis is designed to analyze a predetermined part of the program [175, 170]. The analysis either extracts a result that is valid for all possible contexts in which the fragment may be placed [169], or is designed to analyze the fragment in the context of a whole-program analysis result from a less expensive algorithm [170]. A similar effect may be obtained by explicitly specifying the analysis results for missing parts of the program [115, 175]. Our approach differs in that it monitors the analysis results to dynamically determine which parts of the program it should analyze to obtain the best optimization outcome.

Incremental algorithms update an existing analysis result to reflect the effect of program changes [209]. Our algorithm, in contrast, analyzes part of the program assuming no previous analysis results.

9.8 Conclusion

This paper presents a new incrementalized pointer and escape analysis. Instead of analyzing the whole program, the analysis executes under the direction of an analysis policy. The policy continually monitors the analysis results to direct the incremental analysis of those parts of the program that offer the best marginal return on the invested analysis resources. Our experimental results show that our analysis, when used for stack allocation, usually delivers almost all of the benefit of the whole-program analysis at a fraction of the cost. And because it analyzes only a local region of the program surrounding each allocation site, it scales to handle programs of arbitrary size.

Bibliography

[1] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proc. 9th ECOOP*. LNCS, 1995.

[2] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 207–222, Denver, Colorado, Nov. 1999.

[3] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the Java language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1997.

[4] O. Agesen, J. Palsberg, and M. Schwartzbach. Type inference of SELF: analysis of objects with dynamic and multiple inheritance. *Software—Practice and Experience*, 25(9):975–995, Sept. 1995.

[5] A. Aggarwal and K. H. Randall. Related field analysis. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI)*, pages 214–220, Snowbird, Utah, June 2001.

[6] G. Agrawal. Simultaneous demand-driven data-flow and call graph analysis. Aug. 1999.

[7] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., Reading, MA, second edition, 1986.

[8] A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35:79–111, 1999.

[9] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Programming Language Design and Implementation (PLDI)*, June 1995.

[10] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proc. 21st ACM POPL*, pages 163–173, New York, NY, 1994.

[11] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An introduction to the database programming language Fibonacci. *The VLDB Journal*, 4(3), 1995.

[12] J. Aldrich, C. Chambers, E. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from java programs. In *Proceedings of the 6th International Static Analysis Symposium*, Sept. 1999.

[13] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[14] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In *Proc. 11th ECOOP*, 1997.

[15] C. S. Ananian. Silicon C: A hardware backend for SUIF. Available from http://flex-compiler.lcs.mit.edu/SiliconC/paper.pdf, May 1998.

[16] C. S. Ananian. MIT FLEX compiler infrastructure for Java. Available from http://www.flex-compiler.lcs.mit.edu, 1998-2004.

[17] C. S. Ananian. Static single information form. Master's thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Sept. 1999.

[18] C. S. Ananian. The static single information form. Technical Report MIT-LCS-TR-801, Massachusetts Institute of Technology, 1999. Available from `http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-801.pdf`.

[19] C. S. Ananian and M. Rinard. Data size optimizations for java programs. In *Proceedings of the 2003 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '03)*, June 2003.

[20] C. S. Ananian and M. Rinard. Data size optimizations for java programs. In *2003 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '03)*, San Diego, June 2003.

[21] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[22] A. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[23] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Principles of Programming Languages (POPL)*, January 2003.

[24] D. F. Bacon, S. J. Fink, and D. Grove. Space- and time-efficient implementation of the Java object model. In B. Magnusson, editor, *Proceedings of the 16th European Conference on Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 111–132, Málaga, Spain, June 2002.

[25] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for Java. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, pages 258–268, Montreal, Canada, 1998.

[26] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 324–341, California, 1996.

[27] H. G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978.

[28] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM PLDI*, 2001.

[29] Barendsen and J. E. W. Smetsers. Conventional and uniqueness typing in graph rewrite systems. In *Proceedings of the 13th Conference on the Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, 1993.

[30] W. Beebee and M. Rinard. An implementation of scoped memory for real-time java. In *Proceedings of Embedded Software, First International Workshop, EMSOFT 2001*, Oct. 2001.

[31] W. Beebee and M. Rinard. An implementation of scoped memory for real-time java. Oct. 2001.

[32] W. Beebee, Jr. Region-based memory management for Real-Time Java. MEng thesis, Massachusetts Institute of Technology, September 2001.

[33] W. Beebee, Jr. and M. Rinard. An implementation of scoped memory for Real-Time Java. In *First International Workshop on Embedded Software (EMSOFT)*, October 2001.

[34] M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for linked data structures. In *Proc. 8th ESOP*, 1999.

[35] B. Blanchet. Escape analysis for object oriented languages. Application to Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, Nov. 1999.

[36] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, Sept. 1988.

[37] J. Bogda and U. Hoelzle. Removing unnecessary synchronization in java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, Nov. 1999.

[38] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, Reading, Mass., 2000.

[39] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Springer-Verlag, 1997.

[40] N. Bourbaki. *Theory of Sets*. Paris, Hermann, 1968.

[41] C. Boyapati. Ownership types for safe object-oriented programming. PhD thesis, Massachusetts Institute of Technology. In preparation.

[42] C. Boyapati, R. Lee, and M. Rinard. Safe runtime downcasts with ownership types. Technical Report TR-853, MIT Laboratory for Computer Science, June 2002.

[43] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[44] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL)*, January 2003.

[45] C. Boyapati, B. Liskov, L. Shrira, C. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2003.

[46] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.

[47] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *PLDI 2003*, June 2003.

[48] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[49] D. Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22:251–256, 1979.

[50] M. Budiu, S. Goldstein, M. Sakr, and K. Walker. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *Proceedings of the EuroPar 2000 European Conference on Parallel Computing*. Munich, Germany, Aug. 2000.

[51] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *Proc. 10th International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[52] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[53] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation (PLDI)*, number 6 in 26, pages 278–292, 1991.

[54] C. Chambers. Predicate classes. In *Proc. 7th ECOOP*, pages 268–296, 1993.

[55] R. Chandra, A. Gupta, and J. Hennessy. Data locality and load balancing in COOL. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993. ACM, New York.

[56] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proc. ACM PLDI*, 1990.

[57] R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *Proc. 26th ACM POPL*, pages 133–146, 1999.

[58] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, Nov. 1999.

[59] M. V. Christiansen, F. Henglein, H. Niss, and P. Velschow. Safe region-based memory management for objects. Technical Report D-397, DIKU, University of Copenhagen, October 1998.

[60] M. V. Christiansen and P. Velschrow. Region-based memory management in Java. Master's thesis, University of Copenhagen, May 1998.

[61] D. G. Clarke. Ownership and containment. PhD thesis, University of New South Wales, Australia, July 2001.

[62] D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[63] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proc. 13th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1998.

[64] D. G. Clarke and T. Wrigstad. External uniqueness is unique enough. In *European Conference for Object-Oriented Programming (ECOOP)*, July 2003.

[65] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997. release 1999.

[66] J. C. Corbett. Using shape analysis to reduce finite-state models of concurrent Java programs. *Software Engineering and Methodology*, 9(1):51–93, 2000.

[67] A. Corsaro and D. Schmidt. The design and performance of the jRate Real-Time Java implementation. In *International Symposium on Distributed Objects and Applications (DOA)*, October 2002.

[68] A. Corsaro and D. Schmidt. Evaluating Real-Time Java features and performance for real-time embedded systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, September 2002.

[69] B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In *Handbook of graph grammars and computing by graph transformations, Vol. 1 : Foundations*, chapter 5. World Scientific, 1997.

[70] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th POPL*, 1977.

[71] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proc. 26th ACM POPL*, 1999.

[72] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[73] M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1995.

[74] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM PLDI*, 2001.

[75] B. Demsky and M. C. Rinard. Role-based exploration of object-oriented programs. In *Proceedings of the 2002 International Conference on Software Engineering*, May 2002.

[76] M. Deters and R. Cytron. Automated discovery of scoped memory regions for Real-Time Java. In *International Symposium on Memory Management (ISMM)*, June 2002.

[77] M. Deters, N. Leidenfrost, and R. Cytron. Translation of Java to Real-Time Java using aspects. In *International Workshop on Aspect-Oriented Programming and Separation of Concerns*, August 2001.

[78] D. L. Detlefs, K. R. M. Leino, and G. Nelson. Wrestling with rep exposure. Technical report, DIGITAL Systems Research Center, 1998.

[79] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Technical Report 159, COMPAQ Systems Research Center, 1998.

[80] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *SIGPLAN Notices*, 29(6):230–241, 1994.

[81] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, Aug. 1999.

[82] A. Diwan, K. McKinley, and J. E. B. Moss. Type-based alias analysis. In *Proc. ACM PLDI*, 1998.

[83] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *Proc. 7th International Static Analysis Symposium*, 2000.

[84] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle: Dynamic object re-classification. In *Proc. 15th ECOOP*, LNCS 2072, pages 130–149. Springer, 2001.

[85] E. Duesterwald, R. Gupta, and M. Soffa. A practical framework for interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, Nov. 1997.

[86] J. Ellson, E. Gansner, E. Koutsofios, and S. North. Graphviz. `http://www.research.att.com/sw/tools/graphviz`.

[87] J. Engelfriet. Context-free graph grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages. Vol. III: Beyond Words*, chapter 3, pages 125–213. Springer, 1997.

[88] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In *International Conference on Software Engineering*, pages 449–458, 2000.

[89] M. D. Ernst, Y. Kataoka, W. G. Griswold, and D. Notkin. Dynamically discovering pointer-based program invariants. Technical Report UW-CSE-99-11-02, University of Washington, November 1999.

[90] D. Evans. Static detection of dynamic memory errors. In *Proc. ACM PLDI*, 1996.

[91] R. Familiar. Adaptive role playing. `http://www.ccs.neu.edu/research/demeter/adaptive-patterns/arp-bofam-checked.html`.

[92] S. Feizabadi, W. S. Beebee, B. Ravindran, P. Li, and M. C. Rinard. Utility accrual scheduling with real-time java. In *Proceedings of The First Workshop on Java Technologies for Real-Time and Embedded Systems*, Nov. 2003.

[93] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Principles of Programming Languages (POPL)*, January 1998.

[94] M. Fowler. Dealing with roles.
`http://www.martinfowler.com/apsupp/roles.pdf`, July 1997.

[95] P. Fradet, R. Gaugne, and D. L. Metayer. An inference algorithm for the static verification of pointer manipulation. Technical Report 980, IRISA, 1996.

[96] P. Fradet and D. L. Métayer. Shape types. In *Proc. 24th ACM POPL*, 1997.

[97] P. Fradet and D. L. Métayer. Structured gamma. *Science of Computer Programming, SCP, 31(2-3), pp. 263-289*, 1998.

[98] E. Gamma, R. Helm, R. Johnson, and J. Vlisside. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Mass., 1994.

[99] D. Gay and A. Aiken. Memory management with explicit regions. In *Proc. ACM PLDI*, Montreal, Canada, June 1998.

[100] D. Gay and A. Aiken. Language support for regions. In *Proc. ACM PLDI*, 2001.

[101] F. Gecseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages. Vol. III: Beyond Words*, chapter 1. Springer, 1997.

[102] G. Ghelli and D. Palmerini. Foundations for extensible objects with roles. In *Proc. 6th Workshop on Foundations of Object-Oriented Languages*, 1999.

[103] O. Gheorghioiu. Statically determining memory consumption of real-time Java threads. Master's thesis, Massachusetts Institute of Technology, 2002.

[104] O. Gheorghioiu, A. Sălcianu, and M. C. Rinard. Interprocedural compatibility analysis for static object preallocation. In *Principles of Programming Languages (POPL)*, January 2003.

[105] R. Ghiya and L. Hendren. Is it a tree, a DAG, or a cyclic graph? In *Proc. 23rd ACM POPL*, 1996.

[106] R. Ghiya and L. J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. In *Proc. 8th Workshop on Languages and Compilers for Parallel Computing*, 1995.

[107] R. Ghiya and L. J. Hendren. Putting pointer analysis to work. In *Proc. 25th ACM POPL*, 1998.

[108] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Sun Microsystems, Inc., 2001.

[109] G. Gottlob, M. Schrefl, and B. Roeck. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3), 1994.

[110] D. Grossman. Type-safe multithreading in Cyclone. In *Workshop on Types in Language Design and Implementation (TLDI)*, January 2003.

[111] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proc. ACM PLDI*, 2002.

[112] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[113] C. A. Gunter and J. C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming.* The MIT Press, Cambridge, Mass., 1994.

[114] J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[115] S. Guyer and C. Lin. Optimizing the use of high performance libraries. Aug. 2000.

[116] S. Z. Guyer and C. Lin. An annotation language for optimizing software libraries. In *Second Conference on Domain Specific Languages*, 1999.

[117] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic.* The MIT Press, Cambridge, Mass., 2000.

[118] C. Hauser, C. Jacobi, M. Theimer, B. Welch, and M. Weiser. Using threads in interactive systems: A case study. In *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, Asheville, NC, Dec. 1993.

[119] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI)*, pages 254–263, Snowbird, Utah, June 2001.

[120] L. J. Hendren, J. Hummel, and A. Nicolau. A general data dependence test for dynamic, pointer-based data structures. In *Proc. ACM PLDI*, 1994.

[121] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *Proc. 5th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1991.

[122] G. J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[123] J. Hummel. *Data Dependence Testing in the Presence of Pointers and Pointer-Based Data Structures.* PhD thesis, Dept. of Computer Science, Univ. of California at Irvine, 1998.

[124] J. Hummel, L. J. Hendren, and A. Nicolau. Abstract description of pointer data structures: An approach for improving the analysis and optimization of imperative programs. *ACM Letters on Programming Languages and Systems*, 1(3), Sept. 1993.

[125] J. Hummel, L. J. Hendren, and A. Nicolau. A language for conveying the aliasing properties of dynamic, pointer-based data structures. In *Proc. 8th International Parallel Processing Symposium*, Cancun, Mexico, Apr. 26–29 1994.

[126] S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. 28th ACM POPL*, 2001.

[127] D. Jackson. Alloy: A lightweight object modelling notation. Technical Report 797, MIT Laboratory for Computer Science, 2000.

[128] D. Jackson and J. Chapin. Redesigning air-traffic control: A case study in software design, 2000.

[129] D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. In *International Conference on Software Engineering*, pages 194–202, 1999.

[130] B. Jacobs. Patterns using procedural/relational paradigm. `http://www.geocities.com/tablizer/prpats.htm`.

[131] J. L. Jensen, M. E. Jørgensen, N. Klarlund, and M. I. Schwartzbach. Automatic verification of pointer programs using monadic second order logic. In *Proc. ACM PLDI*, Las Vegas, NV, 1997.

[132] P. Jouvelot and D. K. Gifford. Algebraic reconstruction of types and effects. In *Proc. 18th ACM POPL*, 1991.

[133] N. Klarlund and M. I. Schwartzbach. Graph types. In *Proc. 20th ACM POPL*, Charleston, SC, 1993.

[134] N. Klarlund and M. I. Schwartzbach. Graphs and decidable transductions based on edge constraints. In *Proc. 19th Colloquium on Trees and Algebra in Programming*, number 787 in LNCS, 1994.

[135] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, May 1996.

[136] N. Kobayashi. Quasi-linear types. In *Proc. 26th ACM POPL*, 1999.

[137] J. Korn, Y.-F. Chen, and E. Koutsofios. Chava: Reverse engineering and tracking of Java applets. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, pages 314–325, October 1999.

[138] V. Kuncak, P. Lam, and M. Rinard. A language for role specifications. In *Proceedings of the 14th Workshop on Languages and Compilers for Parallel Computing*, volume 2624 of *Lecture Notes in Computer Science, Springer*, 2001.

[139] V. Kuncak, P. Lam, and M. Rinard. Roles are really great! Technical Report 822, Laboratory for Computer Science, Massachusetts Institute of Technology, 2001.

[140] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *Proc. 29th ACM POPL*, 2002.

[141] C. Lapkowski and L. J. Hendren. Extended SSA numbering: Introducing SSA properties to languages with multi-level pointers. In *Proc. 7th International Conference on Compiler Construction*. LNCS, Mar. 1998.

[142] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley, Reading, Mass., 2000.

[143] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. Research Report 160, Compaq Systems Research Center, November 2000.

[144] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Programming Language Design and Implementation (PLDI)*, June 2002.

[145] T. Lev-Ami. TVLA: A framework for kleene based logic static analyses. Master's thesis, Tel-Aviv University, Israel, 2000.

[146] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, 1981.

[147] Y. Lin and D. Padua. Demand-driven interprocedural array property analysis. La Jolla, CA, Aug. 1999.

[148] B. Liskov and J. M. Wing. A new definition of the subtype relation. *Proc. 7th ECOOP*, 1993.

[149] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Principles of Programming Languages (POPL)*, January 1988.

[150] N. Lynch and F. Vaandrager. Forward and backward simulations – Part I: Untimed systems. *Information and Computation, 121(2)*, 1995.

[151] D. Marinov and R. O'Callahan. Object equality profiling. Submitted to OOPSLA '03, 2003.

[152] A. Møller and M. I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. ACM PLDI*, 2001.

[153] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *Principles of Programming Languages (POPL)*, January 1997.

[154] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[155] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *Proc. 12th ECOOP*, 1998.

[156] T. Onodera and K. Kawachiya. A study of locking objects with bimodal fields. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 223–237, Denver, Colorado, Nov. 1999.

[157] V. Pai, P. Druschel, and W. Zwaenepol. Flash: An efficient and portable web server. In *Proceedings of the Usenix 1999 Annual Technical Conference*, June 1999.

[158] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, Reading, Mass., 1994.

[159] J. Plevyak, V. Karamcheti, and A. A. Chien. Analysis of dynamic structures for efficient parallel execution. In *Workshop on Languages and Compilers for Parallel Architectures*, 1993.

[160] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. In *Communications of the ACM 33(6):668–676*, 1990.

[161] Rational Inc. The unified modeling language. http://www.rational.com/uml.

[162] T. Reenskaug. *Working With Objects.* Prentice Hall, 1996.

[163] J. Reppy. *Higher–order Concurrency.* PhD thesis, Dept. of Computer Science, Cornell Univ., Ithaca, N.Y., June 1992.

[164] T. Reps, S. Horowitz, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the ACM SIGSOFT 95 Symposium on the Foundations of Software Engineering*, Oct. 1995.

[165] J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In *Proceedings of the Symposium in Celebration of the Work of C.A.R. Hoare*, 2000.

[166] W. E. Riddle, J. Sayler, A. Segal, and J. Wileden. An introduction to the dream software design system. volume 2, pages 11–23, July 1977.

[167] M. Rinard. Analysis of multithreaded programs. In *Proceedings of 8th Static Analysis Symposium*, Paris, France, July 2001.

[168] N. Rinetzky and M. Sagiv. Interprocedual shape analysis for recursive programs. In *Proc. 10th International Conference on Compiler Construction*, 2001.

[169] A. Rountev and B. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. Apr. 2001.

[170] A. Rountev, B. Ryder, and W. Landi. Data-flow analysis of program fragments. Sept. 1999.

[171] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations Vol.1.* World Scientific, 1997.

[172] E. Ruf. Effective synchronization removal for Java. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000.

[173] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the SIGPLAN '99 Conference on Program Language Design and Implementation*, Atlanta, GA, May 1999.

[174] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indexes, and accessed memory regions. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000.

[175] R. Rugina and M. Rinard. Design-driven compilation. In *Proc. 10th International Conference on Compiler Construction*, 2001.

[176] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modelling Language Reference Manual.* Addison-Wesley, Reading, Mass., 1999.

[177] J. R. Russell, R. E. Strom, and D. M. Yellin. A checkable interface language for pointer-based structures. In *Proceedings of the workshop on Interface definition languages*, 1994.

[178] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Proc. 23rd ACM POPL*, 1996.

[179] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. 26th ACM POPL*, 1999.

[180] E. Schonberg, J. T. Schwartz, and M. Sharir. An automatic technique for selection of data representations in Setl programs. *Transactions on Programming Languages and Systems*, 3(2):126–143, 1991.

[181] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis problems. In *Program Flow Analysis: Theory and Applications.* Prentice-Hall, Inc., 1981.

[182] M. Sipser. *Introduction to the Theory of Computation.* PWS Publishing Company, 1997.

[183] F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proc. 9th ESOP*, Berlin, Germany, Mar. 2000.

[184] V. C. Sreedhar, M. Burke, and J.-D. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 196–207. ACM Press, 2000.

[185] B. Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, Jan. 1996.

[186] M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI)*, pages 108–120, Vancouver, Canada, June 2000.

[187] R. E. Strom and D. M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Transactions on Software Engineering*, May 1993.

[188] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, January 1986.

[189] A. Sălcianu. Pointer analysis and its applications to Java programs. Master's thesis, MIT Laboratory for Computer Science, 2001.

[190] A. Sălcianu, C. Boyapati, W. Beebee, Jr., and M. Rinard. A type system for safe region-based memory management in Real-Time Java. Technical Report TR-869, MIT Laboratory for Computer Science, November 2002.

[191] A. Sălcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. 2001.

[192] P. F. Sweeney and F. Tip. A study of dead data members in C++ applications. In *Proc. ACM PLDI*, Montreal, Canada, 1998.

[193] W. Thomas. Languages, automata, and logic. In *Handbook of Formal Languages Vol.3: Beyond Words*. Springer-Verlag, 1997.

[194] M. Tofte and L. Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4), July 1998.

[195] M. Tofte and J. Talpin. Region-based memory management. In *Information and Computation 132(2)*, February 1997.

[196] M. Tofte and J. Talpin. Implementing the call-by-value λ-calculus using a stack of regions. In *Principles of Programming Languages (POPL)*, January 1994.

[197] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proc. 11th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1996.

[198] F. Vivien and M. C. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI 2001)*, June 2001.

[199] P. Wadler. Linear types can change the world! In *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, 1990.

[200] D. Walker and G. Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, 2000.

[201] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr. 1991.

[202] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, Nov. 1999.

[203] R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *Proc. 9th International Conference on Compiler Construction*, Berlin, Germany, 2000. Springer-Verlag.

[204] R. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. ACM PLDI*, June 1995.

[205] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture.* ACM, New York, June 1995.

[206] Z. Xu, B. Miller, and T. Reps. Safety checking of machine code. In *Proc. ACM PLDI*, 2000.

[207] Z. Xu, T. Reps, and B. Miller. Typestate checking of machine code. In *Proc. 10th ESOP*, 2001.

[208] P. M. Yelland. Experimental classification facilities for Smalltalk. In *Proc. 6th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1992.

[209] J. Yur, B. Ryder, and W. Landi. Incremental algorithms and empirical comparison for flow- and context-sensitive pointer aliasing analysis. May 1999.

[210] Y. Zhang and R. Gupta. Data compression transformations for dynamically allocated data structures. In *International Conference on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 14–28, Grenoble, France, Apr. 2002. Springer Verlag.

THIS PAGE WAS INTENTIONALLY LEFT BLANK

Chapter 10

Acronyms

BBN	Bolt, Beranek, and Newman
CTAS	Center-Tracon Automation System
Flex	Flexible Compiler Infrastructure for Java developed at MIT
JDK	Java Development Kit
JVM	Java Virtual Machine
MIT	Massachusetts Institute of Technology
OEP	Open Experimental Platform
POSIX	Portable Operating System Interface Standards
RTSJ	Real-Time Specification for Java
SPEC	Standard Performance Evaluation Corporation
SPECjvm98	SPEC Benchmark Suite for the Java Virtual Machine
SPLASH	Stanford Parallel Applications for Shared Memory Benchmark Suite
TCP/IP	Transmission Control Protocol/Internet Protocol
UML	Unified Modeling Language